Fast Startup Internet Congestion Control Mechanisms for Broadband Interactive Applications

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde
eines Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Michael Scharf

geb. in Darmstadt

Hauptberichter:	Prof. Dr.-Ing. Dr. h. c. mult. Paul J. Kühn
Mitberichter:	Prof. Anja Feldmann, Ph. D. (TU Berlin)
	Prof. Dr.-Ing. Andreas Kirstädter
Tag der Einreichung:	6. November 2009
Tag der mündlichen Prüfung:	18. April 2011

Institut für Kommunikationsnetze und Rechnersysteme
der Universität Stuttgart

2011

Kurzfassung

Das Internet kann nie schnell genug sein. Reaktionsschnelligkeit ist und bleibt eine der wichtigsten Eigenschaften von Internet-Anwendungen. Die meisten Anwendungen im Internet nutzen das Transmission Control Protocol (TCP) für die gesicherte Datenübertragung ohne Leistungszusicherungen. TCP ist ein reines Ende-zu-Ende-Transportschichtprotokoll und verwendet Überlastregelungsmechanismen, um die Senderate an die Eigenschaften des Pfades anzupassen. Eine fundamentale Herausforderung für eine derartige Ende-zu-Ende-Überlastregelung ist die Startphase eines neuen Verkehrsflusses: Unmittelbar nach dem Verbindungsaufbau bzw. nach längeren Leerlaufzeiten kann ein Sender nicht einfach eine angemessene Senderate einstellen, da Informationen über den Pfad fehlen. Traditionell verwendet die Überlastregelung von TCP in diesen Fällen die Slow-Start-Heuristik. Dieser Mechanismus funktioniert in vielen Fällen gut, kann allerdings auch zu deutlichen Verzögerungen der Datenübermittlung führen.

Schnellstart-Überlastregelung stellt ein neues Netzparadigma dar, welches zum Ziel hat, diese Unzulänglichkeit zu überwinden, die eine der verbleibenden ungelösten Fragen der Internet-Architektur darstellt. Ein beschleunigter Startvorgang würde insbesondere *breitbandigen interaktiven Anwendungen* nützen. Typische Vertreter dieser Anwendungsklasse sind Web-Anwendungen, welche mit großen Datenmengen operieren, wie z. B. dreidimensionale Inhalte in realen oder virtuellen Welten. Derartige Anwendungen reagieren sensibel auf Latenzen und erfordern eine gesicherte Datenübertragung mit hohen Datenraten und minimalen Verzögerungszeiten.

Die grundsätzliche Herausforderung für jeden Mechanismus zum Starten eines Verkehrsflusses ist die Ermittlung der verfügbaren Datenrate auf dem Pfad. Schnellstart-Mechanismen können inkrementell durch neue Algorithmen innerhalb einer Ende-zu-Ende-Überlastregelung realisiert werden, welche die Datenrate aggressiver als der existierende Slow-Start-Standardmechanismus erhöhen. Dieses Vorgehen riskiert inhärent die Verursachung von Überlast. Eine vielversprechende Alternative ist eine Netzunterstützung durch zusätzliche pfadgebundene Signalisierung zwischen den Endsystemen und den Vermittlungsknoten. Eine derartige Signalisierung ist ein bekannter Mechanismus in Dienstgüte-Architekturen. In letzter Zeit wurden jedoch auch Schnellstart-Überlastregelungsverfahren mit Netzunterstützung entwickelt. Diese Ansätze verwenden zusätzliche Rückmeldungen der Vermittlungsknoten auf dem Pfad. Dennoch sind sie einfach und erfordern keine flussbezogene Zustandshaltung in den Netzkomponenten. Ein Beispiel für diesen Ansatz ist die Quick-Start-TCP-Erweiterung, mit deren Hilfe Endsysteme eine Datenrate anfordern können, die größer als die standardmäßig zulässige Anfangsrate ist. Dies vermeidet den zeitintensiven Slow-Start, falls der Pfad nicht vollständig genutzt ist.

Diese Arbeit untersucht die Realisierung von Mechanismen zur Schnellstart-Überlastregelung. Hierbei wird der Lösungsraum aufgearbeitet, Algorithmen entworfen und verglichen, Implementierungsaspekte in Protokollstapeln betrachtet, sowie der Nutzen und die Konsequenzen bewertet, welche aus der Verwendung in breitbandigen interaktiven Anwendungen resultieren.

Mechanismen zum schnellen Fluss-Start wirken sich sowohl auf das Netz als auch auf Anwendungen aus. Daher behandelt diese Arbeit beide Gesichtspunkte: Die ersten Kapitel führen in paketvermittelte Netze allgemein und speziell deren Ressourcenverwaltung ein und analysieren den Unterschied zwischen einer Ressourcenverwaltung durch Überlastregelung bzw. anderen Verkehrssteuerungsmechanismen. Anschließend werden die Leistungsanforderungen interaktiver Anwendungen betrachtet, sowie existierende Lösungen zur Verbesserung der Reaktionsschnelligkeit. Beide Herangehensweisen zeigen den möglichen Nutzen einer Schnellstart-Überlastregelung auf. Der Hauptteil des Dokuments beinhaltet eine umfassende Analyse, wie schnellere Startvorgänge für Verkehrsflüsse realisiert werden können. Dabei werden die grundlegenden Konzepte und der Lösungsraum für Überlastregelung im Internet aufgearbeitet, die bekannten Lösungsansätze klassifiziert, Erweiterungen und neuartige Kombination vorgeschlagen sowie ungelöste Fragen aufgezeigt.

Aus einer Vielzahl unterschiedlicher Möglichkeiten heraus werden vier TCP-Erweiterungen genauer analysiert: Drei das Ende-zu-Ende-Prinzip beibehaltende Verfahren zur Schnellstart-Überlastregelung unterscheiden sich in der Aggressivität und in der Verwendung von Verfahren zur Verkehrsglättung. Ein weiterer Schwerpunkt ist die Quick-Start-TCP-Erweiterung, welche auf Netzunterstützung basiert. Da Überlastregelung im Internet kein rein algorithmisches Problem darstellt, wurden alle TCP-Erweiterungen in dem Protokollstapel des Linux-Betriebssystems implementiert. Dies ermöglicht Experimente mit einem Protokollstapel auf neustem Stand der Technik. Die durchgeführten Leistungsuntersuchungen basieren sowohl auf Messungen wie auch auf Simulationsstudien mit dem Quelltext des realen Linux-Protokollstapels, deren Ergebnisse gründlich validiert werden. Die Experimente werden noch ergänzt durch die simulative Untersuchung zweier anderer, verwandter Rahmenwerke zur Überlastregelung mit Netzunterstützung, welche im Rahmen von langfristigen Forschungsaktivitäten zur Architektur des künftigen Internet entwickelt wurden.

Diese Arbeit stellt die erste veröffentlichte Vergleichsstudie zu Schnellstart-Überlastregelungsverfahren dar, welche entweder auf Ende-zu-Ende-Mechanismen oder auf Netzunterstützung basieren. Sowohl Simulationsstudien als auch Messungen bestimmen die Leistungssteigerungen, das Risiko der Verursachung von Überlast sowie die Vorteile einer Netzunterstützung. Alle betrachteten TCP-Erweiterungen sind in der Lage, die Transportverzögerungen mittelgroßer Datenmengen zu verringern, insbesondere im Fall breitbandiger Netze mit nicht vernachlässigbaren Übertragungslatenzen. Der erzielbare Leistungsvorteil hängt jedoch von Anwendungscharakteristika ab, so dass nur ausgewählte Anwendungen einen erheblichen Nutzen haben. Die Ergebnisse zeigen auch, dass Ende-zu-Ende-Verfahren nicht notwendigerweise übermäßig aggressiv bzw. unfair sind, falls diese selektiv eingesetzt und entsprechend optimiert werden. Zusätzliche pfadgekoppelte Signalisierung verringert das Risiko von Überlast auf Kosten einer höheren Komplexität. Diese Arbeit zeigt dabei auch, dass netzunterstütze Überlastregelungsverfahren stark von der Richtigkeit der Informationen über die Eigenschaften eines Netzsegments abhängen; fehlerhafte Information kann zu erheblichen Problemen führen.

Ein weiterer Beitrag dieser Arbeit sind verschiedene neue Algorithmen, beispielsweise zur Bewilligung von Quick-Start-Anforderungen in Vermittlungsknoten. Diese zeigen eine bessere Leistungsfähigkeit als bekannte Verfahren, insbesondere bezüglich Fairness, ohne dass hierfür eine deutlicher Mehraufwand notwendig wäre. Darüber hinaus wird auch aufgezeigt, dass das Quick-Start-Protokoll in jedem Fall Heuristiken in den Endsystemen voraussetzt, welche unnötige Anforderungen vermeiden. Eine derartige Heuristik wird vorgeschlagen, und deren Nutzen wird nachgewiesen. Netzunterstütze Überlastregelung erfordert sowohl Veränderungen in den Protokollstapeln der Endsysteme als auch eine zusätzliche Paketverarbeitung in den Vermittlungsknoten. Diese Arbeit weist nach, dass TCP-Erweiterungen für schnelle Fluss-Starts leichtgewichtig und skalierbar sind und mit sehr begrenztem Aufwand realisiert werden können, selbst wenn eine Netzunterstützung erforderlich ist. Die Verwendung von Ende-zu-Ende-Mechanismen ist jedoch wesentlich einfacher. Die dieser Arbeit zugrunde liegenden Entwicklungsarbeiten haben auch verschiedene Implementierungsprobleme aufgezeigt, wie beispielsweise Wechselwirkungen mit der TCP-Datenfluss-Steuerung. Entsprechende Lösungsansätze werden ebenfalls vorgestellt.

Schließlich verdeutlichen zwei Anwendungsstudien die Vorteile von Schnellstart-Überlastregelung und deren Integration in reale Anwendungen: Erstens wird gezeigt, dass ein schneller Start von Datenflüssen die Einhaltung von Leistungszielen auf Anwendungsschicht erleichtern könnte, wie beispielsweise maximale Antwortzeiten. Es wird aufgezeigt, wie ein derartiger Mechanismus in Web-Anwendungen integriert werden könnte. Ein zweites Experiment zeigt mit Hilfe einer prototypischen Anwendung, dass Schnellstart-Überlastregelung von erheblichem Nutzen für Anwendungen ist, die mit dreidimensionalen Umgebungsmodellen operieren. Derartige breitbandige interaktive Anwendungen könnten damit als Wegbereiter für den Einsatz von Schnellstart-Überlastregelung dienen.

Zusammenfassend zeigt diese Arbeit, dass Schnellstart-Überlastregelung einen vielversprechenden und unkomplizierten Mechanismus für die künftige Weiterentwicklung der Internet-Ressourcenverwaltung darstellt, auch wenn weitere Untersuchungen in realen Netzen erforderlich sind, um die resultierenden Folgen auf das Internet als Gesamtsystem zu bewerten.

Abstract

The Internet can never be fast enough. Responsiveness continues to be one of the most important properties of Internet applications. Most Internet applications use the Transmission Control Protocol (TCP) for reliable, best effort transport. TCP is a pure end-to-end transport protocol and uses congestion control in order to adapt the sending rate to the characteristics of a path. A fundamental challenge for any end-to-end congestion control is the flow startup phase: After connection setup or after long idle periods, a sender cannot easily determine an appropriate sending rate due to lack of information about the path. Traditionally, TCP's congestion control uses the Slow-Start heuristic in these cases. This flow startup mechanism works well in many cases, but it can significantly delay data delivery.

Fast startup congestion control is a new networking paradigm that aims at overcoming this limitation, which is one of the remaining open issues of the Internet architecture. Such a speedup would in particular be beneficial for *broadband interactive applications*. Typical representatives of this class of applications are Web applications that deal with voluminous data, such as three dimensional content in real or virtual worlds. These applications are latency-sensitive and require reliable transport with high data rates and minimal transport delays.

The fundamental challenge for any flow startup scheme is the detection of the available bandwidth on the path. Fast startups can be realized incrementally by new end-to-end congestion control algorithms that increase the data rate more aggressively than the standard Slow-Start. This approach inherently risks congestion. A promising alternative would be network support, i. e., additional on-path signaling between the endsystems and the routers. Such signaling is a well-known mechanism in Quality of Service architectures. However, recently several network-supported fast startup congestion control schemes have been developed, too. These approaches use additional feedback information from the routers on the path. Still, they are simple and do not require per-flow state information in the network. One example is the Quick-Start TCP extension. With Quick-Start, endsystems can request for a higher-than-default initial sending rate and avoid the time-consuming Slow-Start if the path is underutilized.

This thesis investigates the realization of fast startup congestion control. It discusses the design space, proposes and compares algorithms, studies implementation issues in network stacks, and evaluates the benefits and implications of its usage in broadband interactive applications.

Fast startup mechanisms affect both the network and applications. Therefore, this document bridges the gap between both aspects. The first chapters introduce packet networks and their resource management and review the differences between congestion control and other traffic management approaches. Subsequently, the performance requirements of interactive applications are addressed, as well as existing solutions to improve their responsiveness. Both approaches reveal the need of fast startup congestion control. The main part of this thesis is a comprehensive analysis how fast startups could be realized. It presents the fundamental concepts and the design space of the Internet congestion control, classifies the known fast startup schemes, proposes new extensions and combinations, and also lists the open issues.

Out of a large number of possibilities, four TCP extensions are analyzed in detail: Three end-to-end fast startup congestion control schemes differ in the aggressiveness and in the use of rate pacing. A further focus is the Quick-Start TCP extension, which uses network support. Since Internet congestion control is not a purely algorithmic problem, all TCP extensions have been implemented in the Linux networking stack so that experiments can be performed with a state-of-the-art stack. The performance evaluation studies are based on both measurements and simulations with real network stack code, and the simulation results are thoroughly validated. The experiments are complemented by simulations studies with two other related network-supported congestion control frameworks that originate from ongoing clean-slate research activities on the architecture of the future Internet.

This thesis is the first published comparative study of end-to-end and network supported fast startup mechanisms. Both simulation and measurement experiments quantify the performance improvement, the risk of congestion, and the benefits of network support. All considered TCP extensions can significantly reduce the transport delay of mid-sized data transfers, in particular over broadband networks with a non-negligible latency. But the performance improvement depends on the application characteristics, and only selected applications indeed benefit significantly. The results reveal that end-to-end fast startup schemes are not necessarily overly aggressive and unfair if they are selectively used and carefully tuned. Additional signaling along the path reduces the risk of congestion at the cost of a higher complexity. This thesis also shows that network-supported congestion control depends on correct information about link characteristics; erroneous information can result in significant problems.

Further contributions of this thesis are several new algorithms, e. g., for the approval for Quick-Start requests in routers. These algorithms outperform the known ones in particular with respect to fairness without resulting in much overhead in routers. Also, it is shown that the Quick-Start protocol crucially depends on heuristics in the endsystems that avoid unnecessary requests. Such a heuristic is proposed, and its usefulness is demonstrated. Network-supported congestion control requires modifications in the protocol stacks of endsystems as well as some additional packet processing in routers. This work shows that fast startup TCP extensions are a lightweight and scalable mechanism that can be implemented with very limited processing overhead even if they use network support. Still, end-to-end mechanisms are much simpler to use and to deploy. The implementation work has also revealed several other realization challenges, such as interactions with the TCP flow control. The corresponding solutions are presented, too.

Finally, two case studies illustrate the benefits of fast startup congestion control and the integration in real applications: First, fast startups could facilitate compliance with application performance targets, such as response time deadlines. This document outlines how this mechanism could be integrated in Web applications. A second experiment shows by proof-of-concept that application dealing with three dimensional world models could indeed significantly benefit from fast startup TCP enhancements. Such broadband interactive applications could thus be an enabler for the deployment of fast startup congestion control mechanisms.

In summary, this work shows that fast startups would be a promising and uncomplicated mechanism for the future evolution of the Internet resource management, even though further experiments in real networks are needed in order to evaluate the implications on the whole Internet.

Contents

Kurzfassung **i**

Abstract **v**

Contents **vii**

List of figures **xi**

List of tables **xv**

Abbreviations and symbols **xvii**

1 Introduction **1**
- 1.1 The Internet and its future evolution . . . 1
- 1.2 Problem statement and contributions of this thesis . . . 2
- 1.3 Thesis structure . . . 4

2 Communication network architectures **7**
- 2.1 Packet networks . . . 7
  - 2.1.1 Fundamentals and terminology . . . 7
  - 2.1.2 Packet network technologies . . . 8
  - 2.1.3 Performance and Quality of Service . . . 9
  - 2.1.4 Network Quality of Service mechanisms . . . 9
- 2.2 Internet . . . 11
  - 2.2.1 Historical evolution . . . 11
  - 2.2.2 Design principles . . . 11
  - 2.2.3 Network architecture . . . 12
  - 2.2.4 Protocols . . . 13
  - 2.2.5 Services . . . 14
  - 2.2.6 Network Quality of Service mechanisms . . . 14
- 2.3 IP-based telecommunication networks . . . 17
  - 2.3.1 Historical evolution . . . 17
  - 2.3.2 Design principles . . . 17
  - 2.3.3 Network architecture . . . 17
  - 2.3.4 Protocols . . . 18
  - 2.3.5 Services . . . 19
  - 2.3.6 Network Quality of Service mechanisms . . . 19

2.4 Future evolution . . . 20
2.4.1 Technological progress and trends . . . 20
2.4.2 Architectural challenges for the Future Internet . . . 20
2.4.3 Research directions . . . 22

3 Performance of broadband interactive applications 23
3.1 Internet and Web applications . . . 23
3.1.1 Classification of applications . . . 23
3.1.2 Web application technologies . . . 24
3.1.3 Performance requirements, metrics, and service level agreements . . . 26
3.1.4 Broadband interactive applications . . . 28
3.2 Existing mechanisms for performance improvement and assurance . . . 29
3.2.1 Server mechanisms . . . 29
3.2.2 Distribution mechanisms . . . 31
3.2.3 Client mechanisms . . . 33
3.3 Performance evaluation methodology . . . 34
3.3.1 Role of performance evaluation . . . 34
3.3.2 Methods for performance modeling, prediction, and measurement . . . 35
3.3.3 Traffic and workload modeling . . . 36
3.3.4 Accurate simulation of real network stacks . . . 38

4 Fast startup congestion control mechanisms 43
4.1 Systematic classification of congestion control methods . . . 43
4.1.1 Terminology and fundamentals . . . 43
4.1.2 Congestion control requirements and design space . . . 47
4.1.3 Classification of end-to-end congestion control methods . . . 49
4.1.4 Classification of network-supported congestion control methods . . . 50
4.1.5 Differences to related mechanisms . . . 53
4.2 State-of-the-art and open issues of Internet congestion control . . . 55
4.2.1 Internet standard solution . . . 55
4.2.2 Interaction with network entities . . . 59
4.2.3 Survey of experimental new algorithms and enhancements . . . 60
4.2.4 Impact of network stack implementations . . . 65
4.2.5 Remaining open and unsolved issues . . . 66
4.3 Fast startups: Definition, motivation, and design principles . . . 70
4.3.1 Definition of fast startup congestion control . . . 70
4.3.2 Motivation and challenges of fast startups . . . 71
4.3.3 Design principles of fast startup congestion control . . . 74
4.4 End-to-end fast startup mechanisms . . . 76
4.4.1 Existing Slow-Start enhancements without speedup . . . 76
4.4.2 Existing optimistic fast startup mechanisms . . . 77
4.4.3 Design of enhanced and new optimistic fast startup schemes . . . 79
4.4.4 Other approaches . . . 82
4.5 Network-assisted fast startup mechanisms . . . 82
4.5.1 Overview of the Quick-Start protocol . . . 82
4.5.2 A new admission control concept: Approval control for Quick-Start . . 84
4.5.3 Design of improved approval control algorithms for Quick-Start . . . 88

4.5.4 Other approaches 92
4.6 Network-controlled congestion control 92
4.6.1 Network control as a clean slate approach 92
4.6.2 Overview of the eXplicit Control Protocol (XCP) 93
4.6.3 Overview of the Rate Control Protocol (RCP) 94
4.6.4 Other approaches 95
4.7 Functional comparison of fast startup schemes 96
4.7.1 Systematic comparison of Quick-Start, XCP, and RCP 96
4.7.2 Other published comparative studies and related work 100

5 Application integration and implementation issues and solutions 103
5.1 Solutions for the interface design 103
5.1.1 Application interfaces 103
5.1.2 Network interfaces 108
5.2 Proposed mechanisms to avoid interactions with flow control 109
5.2.1 State-of-the-art receive window auto-tuning 109
5.2.2 Possible interactions with fast startup congestion control 109
5.2.3 Proposed solution and its implications 111
5.2.4 More disruptive alternative solutions 112
5.3 Realization complexity and feasibility 113
5.3.1 Overview of the implementation work 113
5.3.2 Implementation in endsystems 114
5.3.3 High-speed implementation in network components 117
5.3.4 Lessons learned from the case study 118

6 Performance evaluation 121
6.1 Evaluation methodology and tools 121
6.1.1 Simulation scenarios 121
6.1.2 Measurement setups 124
6.1.3 Experiments in a test network 126
6.2 Simulation tool validation 126
6.2.1 Scenario selection 126
6.2.2 Validation scenario: Response function 127
6.2.3 Validation scenario: Head-of-line blocking 129
6.2.4 Summary of the validation experiments 130
6.3 Study of functional design aspects of fast startups 131
6.3.1 Comparison of the startup behavior 131
6.3.2 Differences in convergence and bandwidth sharing 133
6.3.3 Design and evaluation of new algorithms for Quick-Start 135
6.4 Quantification of the potential performance improvement 141
6.4.1 New analytical model for fast startup performance 141
6.4.2 Comparison of the potential speedup 144
6.4.3 Bandwidth sharing properties 148
6.4.4 Studies with synthetic source-level Web models 152
6.4.5 Studies with trace-based workloads 153
6.4.6 Summary of the performance experiments 156
6.5 Robustness, fairness, and risk 157

6.5.1 Dealing with small buffers . . . 157
6.5.2 Fairness compared to TCP's default congestion control . . . 158
6.5.3 Robustness against imprecise information . . . 160
6.6 Complexity and costs of network support . . . 163
6.6.1 Computational overhead of endsystem functions . . . 163
6.6.2 Computational overhead of router functions . . . 164

7 Applicability case studies **167**
7.1 A new Web performance requirement signaling architecture . . . 167
7.1.1 Congestion control with response time deadlines . . . 167
7.1.2 Proof-of-concept realization . . . 169
7.1.3 Exemplary measurement results . . . 170
7.2 Speedup of 3D visualization applications . . . 171
7.2.1 Network challenges of interactive 3D applications . . . 171
7.2.2 Architecture of an interactive 3D visualization application . . . 172
7.2.3 Exemplary measurement results . . . 173

8 Conclusion **175**
8.1 Summary . . . 175
8.2 Outlook . . . 179

A Appendix: Mathematical background **183**
A.1 Distribution functions . . . 183
A.1.1 Exponential distribution . . . 183
A.1.2 Pareto distribution and variants . . . 183
A.1.3 Log-normal distribution . . . 184
A.2 Analytical lower bound for TCP resequencing delays . . . 184
A.2.1 Scope of the model . . . 184
A.2.2 Model details . . . 184

B Appendix: Documentation of parameters **185**
B.1 Measurement setups . . . 185
B.2 General Linux kernel configuration . . . 185
B.3 Default configuration of the fast startup schemes . . . 186
B.3.1 Jump-Start . . . 186
B.3.2 Initial-Start . . . 186
B.3.3 Mega-Start . . . 186
B.3.4 Quick-Start . . . 186

Bibliography **189**

List of figures

2.1 Characteristics of selected packet network technologies 8
2.2 QoS differentiation functions 9
2.3 QoS assurance functions 9
2.4 Illustration of different QoS signaling architectures 10
2.5 Illustration of the Internet protocol stack 12
2.6 Structure of a TCP segment encapsulated in an IP packet 13
2.7 Comparison of different services delivery principles 14
2.8 Receiver-initiated QoS reservation (e. g., RSVP) 15
2.9 Sender-initiated QoS reservation (e. g., NSIS sender-initiated mode) 15
2.10 Overview of the architecture of 3GPP IMS and ETSI TISPAN 18
2.11 IMS session establishment 20

3.1 Internet application portfolio 24
3.2 Client-server document retrieval by HTTP over TCP 25
3.3 Structure of a Web transaction composed of several requests 26
3.4 Example for a Web Service Level Agreement 28
3.5 Multi-process architecture of a Web server 29
3.6 Classification of alternatives for building scalable Web systems 31
3.7 Fundamental architecture of a two-tier/three-tier Web cluster 31
3.8 Basic structure of a content delivery network 32
3.9 A performance enhancement proxy splitting a TCP connection 32
3.10 Capacity planning workflow 34
3.11 Simple queueing model of a Web server 35
3.12 Example of an open fork/join queueing network 35
3.13 Model for the message exchange in a TCP connection 37
3.14 Observed request and response size distribution in the used workload traces . . 38
3.15 Observed connection IAT distribution in the used workload traces 38
3.16 Architecture of a simulation tool with the Network Simulation Cradle (NSC) . 42

4.1 Congestion in a network caused by competing connections 44
4.2 Pipe model of a three-hop path with different narrow and tight links 44
4.3 Overview of different resource management principles and examples 50
4.4 Simplified Congestion Window evolution of an AIMD algorithm 51
4.5 Chiu/Jain vector diagram showing AIMD's convergence to fairness 51
4.6 Different realization alternatives for explicit feedback from the network 53
4.7 IETF specifications related to TCP's congestion control 57
4.8 Traces of CUBIC 63

4.9 Traces of Compound . . . 63
4.10 Classification of Internet applications . . . 64
4.11 Principle operation of re-ECN . . . 68
4.12 Illustration of a flow startup and a flow restart . . . 70
4.13 Scope of fast startup congestion control . . . 71
4.14 Classification of applications that benefit from fast startup schemes . . . 73
4.15 Cross-layer signaling of application and network characteristics . . . 73
4.16 Overview of the design space of flow startup mechanisms . . . 75
4.17 Main phases of a fast startup mechanism . . . 76
4.18 Modified Jump-Start with upper bounds for data rate and outstanding data . . . 80
4.19 Initial rate of Jump-Start in the workload traces . . . 80
4.20 Quick-Start signaling during TCP connection setup . . . 83
4.21 New TCP functions required by Quick-Start . . . 84
4.22 New IP functions required by Quick-Start . . . 84
4.23 Classification of different approval control algorithms . . . 86
4.24 Illustration of the Quick-Start approval control and recent request storage . . . 87
4.25 Quick-Start approval control by the “target algorithm” . . . 89
4.26 Illustration of the approval control by the “fair algorithm” . . . 90
4.27 RTT estimation method for Quick-Start . . . 90
4.28 Operation of XCP and RCP with a new congestion header . . . 93

5.1 Explicit fast startup control . . . 104
5.2 Implicit fast startup control . . . 104
5.3 Querying fast startup parameters from a network information service . . . 105
5.4 Using signaling in order to obtain a rate recommendation . . . 105
5.5 Explicit application interface example . . . 107
5.6 Early activation of Quick-Start . . . 107
5.7 Late activation of Quick-Start . . . 107
5.8 Example for the manual command line configuration of the interface capacity . 108
5.9 Illustration of the difference of congestion control vs. flow control . . . 109
5.10 Illustration of receive window auto-tuning that assumes a Slow-Start . . . 109
5.11 Example for the receive window auto-tuning used by Linux . . . 110
5.12 Illustration of the interaction between a fast startup and flow control . . . 110
5.13 Message sequence chart of the workaround to prevent interactions . . . 112
5.14 Explanation why the RWND could be ignored during the flow startup . . . 113
5.15 Rate pacing with a constant timer granularity . . . 114
5.16 Simplified Jump-Start sender state engine . . . 115
5.17 Simplified Quick-Start sender state engine . . . 115
5.18 Illustration of the Quick-Start TCP implementation in the Linux network stack . 116
5.19 Illustration of a Quick-Start implementation in a network processor . . . 117

6.1 Workflow of simulations with real network stack code . . . 122
6.2 Dumb-bell simulation topology with a central bottleneck . . . 122
6.3 Alternative dumb-bell simulation topology with different RTTs . . . 122
6.4 Quick-Start dispatcher in the “Wireshark” tool that displays traces . . . 125
6.5 Experimental lab setup with several computers . . . 125
6.6 Illustration of the experimental path used in some experiments . . . 125
6.7 Setup of the validation scenario that studies the response function . . . 126

6.8 Setup of the validation scenario for Head-of-Line blocking 126
6.9 Simulated Reno response functions vs. models and measurements 127
6.10 Simulated CUBIC response functions vs. models and measurements 127
6.11 Simulated Compound response functions vs. models and measurements 128
6.12 Runtime performance comparison of the different simulation tools 128
6.13 Comparison of the impact of HOL in simulation and measurements 130
6.14 Response time distribution resulting from HOL 130
6.15 Fast startup and fast restart of different end-to-end mechanisms 131
6.16 Fast startup and fast restart of different network-supported mechanisms 131
6.17 Behavior of competing flows using Slow-Start without and with RED 133
6.18 Behavior of competing flow using Jump-Start or Mega-Start 133
6.19 Behavior of competing flows using Quick-Start with limited buffer 134
6.20 Behavior of competing flows using XCP or RCP with unlimited buffer 134
6.21 Client-server communication with early QS activation 135
6.22 Quick-Start testbed with different router implementations 135
6.23 Quick-Start with different SST adaptation strategies on a 10 Mbit/s path 136
6.24 Quick-Start with different SST adaptation strategies on a 100 Mbit/s path 136
6.25 Quick-Start on an emulated path with several routers without cross-traffic . . . 137
6.26 Quick-Start on an emulated path with several routers with cross-traffic 137
6.27 Quick-Start approval control with the target algorithm using the peak or EWMA rate estimator 138
6.28 Quick-Start approval control with the fair or optimistic algorithm in combination with the peak rate estimator 138
6.29 Quick-Start approval control load test 139
6.30 Efficiency of different approval control algorithms 140
6.31 Fairness of different approval control algorithms 140
6.32 Simplified path model 142
6.33 Fast startup with rate pacing 142
6.34 Benefit of Jump-Start as a function of the transfer size 145
6.35 Benefit of other end-to-end schemes as a function of the transfer size 145
6.36 Benefit of Quick-Start as a function of the transfer size 145
6.37 Benefit of network-controlled schemes as a function of the transfer size 145
6.38 Relative performance improvement of Jump-Start as a function of the RTT . . . 147
6.39 Relative performance improvement of Quick-Start as a function of the RTT . . 147
6.40 Benefit of selected schemes for higher path capacities 147
6.41 Measured speedup on a real 1 Gbit/s path with different QS request rates 147
6.42 Impact of an increased load on Jump-Start 149
6.43 Impact of an increased load on other end-to-end schemes 149
6.44 Impact of an increased load on Quick-Start 150
6.45 Impact of an increased load on XCP/RCP 150
6.46 Load dependency of different QS approval control algorithms 150
6.47 Benefit of activating Quick-Start only for larger transfers 150
6.48 Comparison of the measured page download times for one user 153
6.49 Measurement of the impact of an increased load 153
6.50 Distribution of the epoch time duration for end-to-end fast startups 154
6.51 Distribution of the epoch time duration for network-supported fast startups . . . 154
6.52 Breakdown of the performance depending on transfer sizes (load 6 %) 155

6.53 Breakdown of the performance depending on transfer sizes (load 35 %) 155
6.54 Speedup vs. packet loss for end-to-end schemes 156
6.55 Speedup vs. packet loss for network-supported schemes 156
6.56 Influence of the buffer size on end-to-end schemes (load 35 %) 157
6.57 Influence of the buffer size on Quick-Start (load of 35 %) 157
6.58 Mixed usage of default and end-to-end fast startup congestion control 159
6.59 Mixed usage of default and Quick-Start congestion control 159
6.60 Impact of capacity estimation errors on Quick-Start 161
6.61 Impact of capacity estimation errors on XCP and RCP 161
6.62 Impact of Quick-Start capacity overgranting over the experimental path 162
6.63 Server-side processing effort in the Linux stack obtained by kernel profiling . . 162
6.64 Setup of the load test for Quick-Start routers 164
6.65 Delay distribution of packets through a router under load 164

7.1 HTTP-based signaling of response time requirements 168
7.2 Web application with deadline compliance extensions 169
7.3 Screenshot of the enhanced “Firefox” Web browser 169
7.4 Compliance to a signaled response time deadline for a single object 170
7.5 Influence of the deadline on the “OpenStreetMap” Web page loading time . . . 170
7.6 Client-server architecture of an interactive 3D visualization application 173
7.7 Traces from a 3D visualization tool with Slow-Start vs. Jump-Start 174
7.8 Traces from a 3D visualization tool with Slow-Start vs. Quick-Start 174

8.1 Classification of the congestion control mechanisms considered in this work . . 176

List of tables

2.1 Comparison of the service classes defined by IETF, IEEE, and ITU-T 16

3.1 Classification of network stack simulation approaches 40

4.1 Congestion control design space 49
4.2 Design space of network-supported congestion control 52
4.3 Classification and comparison of important TCP congestion control variants . . 61
4.4 Known characteristics of popular TCP stacks 65
4.5 General transport protocol design space 69
4.6 Arguments for and against fast startup congestion control 72
4.7 Comparison of selected end-to-end Slow-Start enhancements 76
4.8 Comparison of selected fast startup TCP enhancements 92
4.9 Comparison of network-supported congestion control schemes 97
4.10 Reported implementations of network-supported congestion control schemes . 101

5.1 Proposed fast startup application interface extensions 106
5.2 Comparison of the implementation complexity of fast startup schemes 116

6.1 Response time equations for different startup schemes 144
6.2 Important parameters of the used Web workload models 152
6.3 Consequences of false capacity assumptions in network components 160
6.4 Quick-Start implementation performance characteristics 165

8.1 Summary of the properties of fast startup schemes 177

Abbreviations and symbols

Abbreviations

3D	Three-dimensional	167
3GPP	3rd Generation Partnership Project	17
AAA	Authentication, Authorization, Accounting	16
ABR	Available Bit Rate	96
ADSL	Asymmetric Digital Subscriber Line	8
AIMD	Additive Increase Multiplicative Decrease	50
AJAX	Asynchronous JavaScript and XML	25
ALG	Application Level Gateway	12
API	Application Programming Interface	25
AQM	Active Queue Management	59
AS	Application Server	18
ASIC	Application-Specific Integrated Circuit	117
ATM	Asynchronous Transfer Mode	96
BDP	Bandwidth-Delay Product	8
BGP	Border Gateway Protocol	13
BSD	Berkeley Software Distribution	40
BTC	Bulk Transfer Capacity	45
CA	Congestion Avoidance	56
CCDF	Complementary Cumulative Distribution Function	38
CCID	Congestion Control Identifier	14

CDN	Content Delivery Network	31
CPU	Central Processing Unit	117
CSCF	Call Session Control Function	18
CWND	Congestion Window	50
DBMS	Database Management System	30
DCCP	Datagram Congestion Control Protocol	14
DNS	Domain Name System	14
DoS	Denial-of-Service	19
DPI	Deep Packet Inspection	17
DSCP	DiffServ Code Point	16
DUT	Device Under Test	36
ECN	Explicit Congestion Notification	59
ERP	Enterprise Resource Planning	26
ETSI	European Telecommunications Standards Institute	17
EWMA	Exponentially Weighted Moving Average	86
FCFS	First Come First Served	10
FIFO	First In First Out	10
FPGA	Field Programmable Gate Array	101
GUI	Graphic User Interface	24
HOL	Head-of-Line Blocking	74
HSS	Home Subscription Server	18
HTML	Hypertext Markup Language	24
HTTP	Hypertext Transfer Protocol	14
IAT	Inter-Arrival Time	36
ICMP	Internet Control Message Protocol	14
IEEE	Institute of Electrical and Electronics Engineers	16
IETF	Internet Engineering Task Force	3
IKR	Institut für Kommunikationsnetze und Rechnersysteme	40

IMS	IP Multimedia Subsystem	18
IP	Internet Protocol	1
IPTV	Internet Protocol Television	19
IS	Initial-Start	78
ISO	International Organization for Standardization	12
ISP	Internet Service Provider	13
IT	Information Technology	20
ITU	International Telecommunications Union	16
JS	Jump-Start	79
LAN	Local Area Network	8
MAC	Media Access Control	62
MAN	Metropolitan Area Network	8
MBAC	Measurement-Based Admission Control	54
MPLS	Multiprotocol Label Switching	17
MS	Mega-Start	82
MSS	Maximum Segment Size	13
MTU	Maximum Transmission Unit	13
NAT	Network Address Translation	15
NGN	Next Generation Network	4
NNI	Network Network Interface	18
NS-2	Network Simulator version 2	40
NSC	Network Simulation Cradle	41
NSIS	Next Steps in Signaling	15
OAM	Operations, Administration, and Management	11
OS	Operating System	30
OSI	Open Systems Interconnection	12
OSM	OpenStreetMap	167
P2P	Peer-to-Peer	14

PC	Personal Computer	124
PCN	Pre-Congestion Notification	65
PDF	Policy Decision Function	10
PEF	Policy Enforcement Function	10
PEP	Performance Enhancement Proxy	33
PHB	Per Hop Behavior	16
PPS	Packets Per Second	164
PS	Processor Sharing	148
PSTN	Public Switched Telephone Network	17
QoS	Quality of Service	4
QS	Quick-Start	82
RCP	Rate Control Protocol	3
RED	Random Early Detection	59
REST	Representational State Transfer	25
RFC	Request for Comment	5
RPC	Remote Procedure Call	25
RSVP	Resource Reservation Protocol	15
RTO	Retransmission Timeout	56
RTT	Round-Trip Time	1
RWND	Receive Window	56
SACK	Selective Acknowledgment	41
SBC	Session Border Controller	19
SCTP	Stream Control Transmission Protocol	14
SDL	Specification and Description Language	35
SDP	Session Description Protocol	19
SIP	Session Initiation Protocol	18
SLA	Service Level Agreement	9
SOAP	Simple Object Access Protocol	25

SQL	Structured Query Language	30
SS	Slow-Start	56
SST	Slow-Start Threshold	56
SURGE	Scalable Uniform Resource Locator Reference Generator	36
T/TCP	Transaction TCP	82
TCP	Transmission Control Protocol	1
TFRC	TCP Friendly Rate Control	64
TISPAN	Telecommunications and Internet Converged Services and Protocols for Advanced Networks	18
UDP	User Datagram Protocol	13
UE	User Equipment	18
UML	Unified Modeling Language	35
UNI	User Network Interface	18
URI	Uniform Resource Identifier	14
VoIP	Voice-over-IP	14
WAN	Wide Area Network	8
WWW	World Wide Web	1
XCP	eXplicit Control Protocol	3
XML	Extensible Markup Language	25

Symbols

$\mathbf{A}$	Routing matrix	46
a_{QS}	Unused bandwidth available for Quick-Start	88
a_{RCP}	Feedback calculated by the RCP scheme	95
a_{XCP}	Feedback calculated by the XCP scheme	94
B	Buffer size (in packets)	59
b	Instantaneous queue length	95
b_{persist}	Persistent queue length	94
BTC	Bulk Transfer Capacity	45
c	Assumed link capacity	88
d	Measured Round-Trip Time	58
d_{avg}	Average Round-Trip Time	94
d_{max}	Maximum Round-Trip Time	62
$\mathbf{E}$	Epoch in a connection vector of the form (a_i, b_i, t_i) or $(a_i, t_{a,i}, b_i, t_{b,i})$	37
$F(\cdot)$	Distribution function	183
$f(\cdot)$	Probability density function, or placeholder for a function	46
FI	Fairness index	48
G	Throughput	45
h	Link price function	46
H	Entry in a ring buffer for storage of information	88
HZ	Interrupt frequency of the operating system (e. g., Linux)	114
i, j, k	Index variables	24
K_{data}	Upper limit on the amount of data played out by the Jump-Start scheme	81
k_{pareto}	Location parameter of the pareto distribution	183
K_{rate}	Upper limit on the initial sending rate used by the Jump-Start scheme	81
K_{pareto}	Maximum value of the truncated pareto distribution	183
L	Maximum Segment Size	57
M	Accumulated amount of data tranfered in Slow-Start	142

m	Mean value	148
MTU	Maximum Transmission Unit	57
n	Number of entities	48
N_{chunk}	Chunk size during rate pacing	115
N_{hops}	Number of hops on a path	45
N_{hosts}	Number of hosts (either clients or servers)	123
N_{pacing}	Number of segments sent during the rate pacing phase	80
N_{peak}	Number of samples used by the peak estimator	86
N_{queued}	Number of queued segments during Head-of-Line blocking	184
N_{rtx}	Number of segments that are retransmitted after the rate pacing phase	80
N_{timer}	Number of timers during a rate pacing phase	115
$\mathbf{p}$	Link price	46
p	Packet loss probability	58
p_{approv}	Quick-Start request approval probability	151
$\mathbf{q}$	Aggregated link price	46
Q	Initial sending rate at transport layer	81
q	Initial sending rate at network layer	83
q_{approved}	Granted rate of a Quick-Start approval control	88
q_{fair}	Rate determined by a Quick-Start approval control (fair algorithm)	90
$q_{\text{optimistic}}$	Rate determined by a Quick-Start approval control (optimistic algorithm)	89
Q_{req}	Requested sending rate at transport layer	81
q_{req}	Requested sending rate at network layer	83
q_{target}	Rate determined by a Quick-Start approval control (target algorithm)	88
$\mathbf{r}$	Link capacity vector	46
r	Link or path capacity	45
R	TCP path capacity	141
r_{A}	Link capacity of the access links in the dumb-bell topology	123
r_{C}	Link capacity of the central bottleneck in the dumb-bell topology	123

s	Application data to be transfered	81
t	Time	61
t_0	Start time	90
T_{det}	Packet loss detection time	184
T_{epoch}	Duration of an epoch	154
T_{FCT}	Flow completion time	26
T_{o}	Duration of the retransmission timeout	58
T_{page}	Page download time	26
T_{react}	Reaction time	26
T_{resp}	Response time	26
$T_{\text{resp,min}}$	Minimum response time	148
T_{round}	Duration of a Slow-Start round	142
T_{target}	Response time target	168
T_{trans}	Transaction time	26
T_{hol}	Mean waiting time due to Head-of-Line blocking	184
u	Used bandwidth	86
U	Utility function of an application	24
v	Available bandwidth	45
$\mathbf{V}$	Connection vector consisting of epochs	37
w	Initial Congestion Window	56, 110
W	Congestion Window	50
W_{max}	Maximum Congestion Window	58
W_{playout}	Maximum Congestion Window during the rate pacing phase	115
W_{qs}	Quick-Start Congestion Window	83
W_{recov}	Congestion Window after error recovery	80
$\mathbf{x}$	Sending rate vector	46
x	Sending or data rate	24
$\mathbf{y}$	Link rate vector	46

Z	Ramdom variable	183
z	Variable	183

α_{ewma}	Weight factor of the exponential moving average estimator	86
α_{incr}	Window increase factor	50
α_{pareto}	Shape factor of the pareto distribution	38
α_{RCP}	Weight factor of the RCP scheme	95
α_{XCP}	Weight factor of the XCP scheme	94
β_{decr}	Window decrease factor	50
β_{RCP}	Weight factor of the RCP scheme	95
β_{XCP}	Weight factor of the XCP scheme	94
γ	Slow-Start acceleration factor	142
Γ	Data transfer time	142
Δ	Control interval duration	86
δ	Inter-Arrival Time	129
ε	Processing time overhead	129
η	Number of segments acknowledged in one received ACK	58
θ	Threshold up to which a link is considered underutilized	88
Θ	Acceptable unfairness threshold of a Quick-Start approval control	90
κ	Index of the last round in a flow startup	142
$\mathscr{L}$	Set of links	46
μ_{lognorm}	Parameter of the log-normal distribution	184
ν	Index of the last round in a flow startup	142
Ξ	Maximum bandwidth granted by the approval control per second	139
ρ	Utilization	45
$\mathscr{S}$	Set of sources	46
σ_{lognorm}	Parameter of the lognormal distribution	184
τ	Minimum Round-Trip Time	58

χ	Activation threshold for Quick-Start	91
ψ	Index of the last round in a flow startup that is completely used	142
Ω	Size of a ring buffer for the storage of information	87
Ω_{opt}	Optimal size of a ring buffer for the storage of information	88
ω	Waiting time of a segment	184

Nomenclature:

x denotes a scalar, $\mathscr{X}$ a set, $\mathbf{x}$ a vector or a matrix, and $x(\cdot)$ a function

Data rates offered by a link layer techology are generally refered to by a lower case symbol, whereas data rates at transport layer are identified by symbols in capital letters.

This document uses the symbols “KiB” for *kibibyte* (1024 B) and “Mi” for *mebibyte* (1024 · 1024 B), which are standardized by the Institute of Electrical and Electronics Engineers (IEEE).

1 Introduction

1.1 The Internet and its future evolution

The Internet has revolutionized the computer and communications world. Originating from an academic and military community, the Internet was never foreseen to become the ubiquitous and commercial global network that it is today. And it is still evolving rapidly. The most important Internet service is the *World Wide Web* (WWW), which is changing from a simple client-server browsing infrastructure to a complex medium for multimedia content, personalized services (*Web 2.0*), and *Rich Internet Applications* that basically behave like locally installed software. Therefore, fast Internet access is a crucial prerequisite for the future information society.

Computer networks traditionally use connectionless store-and-forward packet switching without resource reservation. As a consequence, congestion, i. e., temporary overload of network components, is an inevitable effect. The Internet globally interconnects computer networks by the *Internet Protocol* (IP) and thus has to cope with such overload situations. *Congestion control* mechanisms detect and react to congestion in order to minimize its impact. Congestion control is an integral part of the *Transmission Control Protocol* (TCP), which is the default transport protocol for reliable, elastic traffic in the Internet. TCP is used by most Internet applications.

Since TCP is a pure end-to-end protocol, the congestion control mechanisms must continuously probe the available bandwidth on the path in order to adapt the sending rate. These algorithms are challenged by paths with a large available bandwidth and/or delay. In a network spanning the globe there are inherent minimum communication delays, in particular if the communication path traverses broadband long-distance links or cellular networks. If the *Round-Trip Time* (RTT) is not negligible, TCP's algorithms have a significant influence on the user-perceived performance of networked applications.

Over the years, more and more efficient congestion control methods have been developed and integrated into TCP. However, after the connection setup or after long idle periods it is difficult to determine an appropriate sending rate due to lack of information about the path. Traditionally, the TCP congestion control uses the *Slow-Start* heuristic in these cases, but this is a time-consuming process that may require many RTTs until an appropriate sending rate is reached. This gap between increasing bandwidths but constant delays is particularly critical for emerging *broadband interactive applications*, which are delay-sensitive applications that interactively exchange potentially large amounts of data. They are most likely to be realized as Web applications with direct user interactions and a client-server system architecture.

The existing TCP congestion control methods have not been designed for such latency-sensitive applications. The Slow-Start can significantly delay data delivery and thus reduce the utility of broadband interactive applications. The flow startup is also one of only few situations in which even advanced TCP congestion control algorithms have a suboptimal performance and there is still room for improvement. This has motivated the design of new congestion control mechanisms that are optimized for interactive usage. This new class of algorithms and protocols can be summarized by the term *fast startup congestion control*. Potential solutions range from incremental TCP enhancements to completely new congestion control frameworks that would require a revision of the Internet architecture. Yet, designing a new flow startup scheme is a difficult problem, and several open research issues have not been completely solved so far.

1.2 Problem statement and contributions of this thesis

This thesis motivates the use of fast startup congestion control, discusses the complete design space, proposes and compares algorithms, studies implementation issues in network stacks, and evaluates the benefits and implications of its usage in broadband interactive applications.

Fast startups are a new mechanism and currently not widely used in the Internet. This raises the question: *How could fast startups be realized, and what would be the consequences?* Answering this question is non-trivial and has to address three fundamental challenges:

1. *Detection of space capacity*: Congestion control must estimate the available bandwidth on the path through the network. No method can instantaneously determine this information. A congestion control scheme can either use *probing*, but this approach inherently risks congestion. Alternatively, there could be *signaling* between the endsystems and the network. While signaling along the path is a well-known mechanism to realize Quality of Service, new lightweight signaling could also enhance the congestion control without requiring per-flow state in the network. Both methods have advantages and drawbacks, which have hardly been comprehensively compared so far.
2. *Trade-off between performance improvement and risk of congestion*: Any flow startup scheme can cause congestion. Even if it was possible to determine the load of all network components on a path, this information would already be out-dated once it would reach the source of a flow. Whatever flow startup scheme is used, the load situation on a resource may have completely changed when the flow indeed arrives. As a result, one has to weight the aggressiveness of a startup scheme against the increased risk of congestion. The current Internet design philosophy is to be conservative, but it is more and more argued that a more aggressive scheme could be feasible.
3. *Internet performance evaluation challenges*: The Internet and its protocols are very heterogeneous and complex. This makes it extremely difficult to forecast the impact of new protocol mechanisms before they actually get deployed.

This thesis addresses these challenges as follows: First, *different classes of fast startup congestion control* mechanisms are analyzed and compared. They include both existing and new end-to-end solutions that have been designed or enhanced by the author, as well as network-supported schemes, which use additional signaling to overcome the lack of information. Second, *extensive experiments* have been performed in order to study the trade-off between speedup and packet loss. A specific focus is the transaction-oriented communication of Web-like applications that could benefit most from new flow startup schemes. Third, the performance evaluation is based on a mix of analytical models, simulations, and some real-world measurements. Instead of simplified models, this thesis uses *simulations with real network stack code*. Most considered protocol mechanisms have been implemented in real network stacks in order to obtain realistic results. Also, a significant effort is spent on the validation of the simulation results against testbed measurements.

This thesis is the first *comparative study* of new end-to-end and network-supported fast startup congestion control schemes and includes the following novel contributions:

- The author has designed new or *enhanced end-to-end fast startup algorithms* for TCP. As an alternative, the Quick-Start extension is considered, which is an experimental network-supported fast startup scheme specified by the *Internet Engineering Task Force* (IETF). The author proposes several *new algorithms for Quick-Start*. In particular, this thesis shows that the resource management in routers can either follow the oversubscription or the bandwidth pooling principle, and both solutions are compared. The novel approval control algorithms outperform existing proposals, in particular with respect to fairness.
- This thesis studies the required TCP enhancements with new *implementations in real network stacks*. In particular, the first reported full implementation of the Quick-Start TCP extension has been realized as a part of this work. The author has also first published results on the complexity and implementation challenges of this as well as other fast startup mechanisms. For instance, the author has shown that there are *interactions of fast startup mechanisms and the TCP flow control* and proposes a backward-compatible solution to this problem, which has been overlooked by all related work.
- The implications of network-supported congestion control are comprehensively discussed, including the proposal of a *precise terminology* and a complete analysis of *open issues*. This thesis compares Quick-Start TCP with two other well-known network-supported congestion control schemes: *eXplicit Control Protocol* (XCP) and *Rate Control Protocol* (RCP). Despite significant research efforts in the context of XCP and RCP, the similarities and differences to Quick-Start are not addressed by any other published work.
- The *performance assessment of different solutions* compares both end-to-end and network supported schemes and is based on new analytical models, simulation results, and measurement data. Thereby, this study is unique. Both for Quick-Start TCP as well as for other end-to-end fast startup schemes there are no other published results that have been obtained with a full-featured TCP implementation under realistic constraints. The experiments in this work are the first ones that use bi-directional, application-limited transfers that are typical for client-server applications.
- Several fast startup variants require *new interfaces between applications and the transport layer*. Such interfaces are designed in this thesis, and their usage is demonstrated by proof-of-concept experiments with real applications.

Internet research is not a pure academic topic and must also consider real-world constraints and non-technical issues. This thesis bridges the gap between fundamental theoretical work and practical relevance. An important, yet not entirely scientific contribution of this work is the proof-of-concept that fast startup congestion control is indeed feasible with limited implementation complexity. The considered fast startup TCP extensions have been prototyped in real network stacks and have been thoroughly tested under realistic constraints. It is good news and an important, albeit not entirely scientifically interesting result that only few hard problems have been identified – and solved. These findings have also contributed to the ongoing discussions on the future evolution of Internet transport protocols.

The main results of this work have been published in the several peer-reviewed papers [199, 203, 204, 205, 207] as well as in other closely related publications of the author [198, 200, 202, 206]. The author is also a main contributor to several other peer-reviewed papers [84, 181] and a comprehensive survey document [172], which are all closely related to this thesis. Other peer-reviewed publications broadly within the scope of this work include the references [195, 196, 162, 197, 238, 116]. The protocol implementations have partly been realized in student projects [253, 223, 226, 180] under guidance of the author.

1.3 Thesis structure

This thesis deals with novel transport protocol mechanisms that are located on the boundary between applications and the network. Fast startup congestion control cannot be discussed as a pure transport layer problem only, but must be analyzed in the context of network and application architectures. This thesis addresses both the network and the application implications of fast startup congestion control and is therefore structured in the following chapters:

Chapter 2 introduces IP-based network architectures. Both the Internet as well as other IP-based network architectures are considered. As congestion control has both similarities and fundamental difference to network *Quality of Service* (QoS) mechanisms, a specific focus of this chapter is the potential realization of such traffic control mechanisms, for instance in *Next Generation Networks* (NGNs). But this chapter also argues why the future Internet evolution will not necessarily be based on NGN technologies, and it outlines how enhanced congestion control mechanisms could continue to be an alternative in the *Future Internet*.

Chapter 3 deals with the performance of interactive applications, in particular in the WWW. It shows why responsiveness is a key user requirement, in particular for *broadband interactive applications*. The chapter reviews the state-of-the-art techniques for the realization of delay-sensitive interactive applications and shows that all known techniques have shortcomings as well, which could be overcome by fast startup congestion control. A further focus is the evaluation methodology for interactive applications. Suitable modeling techniques both for network and application aspects are reviewed. Finally, this chapter motivates and introduces the method of *simulation with real network stack code*, which is used throughout this work.

The purpose of Chapter 4 is to comprehensively discuss fast startup congestion control mechanisms. It starts by an introduction of the fundamental concepts of congestion control and then surveys the current realization in the Internet, which is a multi-faceted topic. The major part of the chapter presents the design space of fast startup congestion control and reviews the known proposals. The survey starts by end-to-end modifications of the Slow-Start. In addition to just increasing the initial window, a promising approach is the *Jump-Start* scheme that uses rate pacing with a high initial sending rate that depends on the application sending pattern. Next, different network supported approaches are analyzed. A specific focus is the *Quick-Start TCP extension*, which queries the routers along the path in order to start a data transfer with a high initial sending rate. As the algorithms used by the routers have a crucial impact on the performance, the published algorithms are analyzed and new extensions are proposed. The author also suggests a new fast startup scheme that combines the principles of Jump-Start and Quick-Start. Finally, other related clean-slate frameworks are presented, such as XCP and RCP. The chapter concludes with a discussion of the open issues of such network support.

Chapter 5 discusses realization aspects of fast startup congestion control, which are hardly addressed by other related work. The first considered aspect is the design of new interfaces between applications and the network stack, which could for instance be used to activate a fast startup. Second, it is shown that fast startup congestion control may require modifications in the TCP flow control and in the receive buffer allocation strategies. And third, the realization complexity of end-to-end and network-supported schemes is compared. Both classes of fast startup schemes have been implemented in the network stack of the Linux operating system, which is a state-of-the-art TCP/IP stack that is widely used in experimental research. The results confirm that end-to-end fast startup schemes only require small extensions in a state-of-the-art network stack, even if they use rate-pacing. Furthermore, it is demonstrated that network-supported mechanism such as Quick-Start can also be implemented with relatively small overhead both in endsystems and routers. Finally, this chapter briefly summarizes the findings of another case study with a hardware-supported Quick-Start router implementation, which has proved that the processing of the signaling is possible with full line speed in high-speed network components.

In Chapter 6, the benefits and risks of several fast startup congestion control alternatives are extensively studied by analytical analysis, simulation studies, and testbed measurements. The author has placed emphasis on the validation of the simulation results by comparison to analytical calculations and testbed measurements, as far as this is possible. In order to study the usefulness of fast startup congestion control, experiments are performed in different application and network scenarios. On the one hand, the results confirm previously known results by more realistic experiments: Fast startup congestion control can significantly reduce the transfer times of mid-sized data transfers, in particular if the path has a large available bandwidth and if the RTT is larger than 100 ms. The potential speedup depends on the realization of the fast startup, but it can easily be of the order of several hundred percent. The benefit also depends very much on the application workload characteristics, and is not necessarily very large for existing Web applications. On the other hand, there are several new findings: The experiments show that the performance of Quick-Start depends on the interplay between the algorithms in the endsystems and in the routers. In order to be useful, intelligent activation and usage strategies are required. Furthermore, it is shown that other network-supported schemes such as XCP or RCP are much less robust than Quick-Start and crucially depend on accurate information about link capacities. Finally, the processing overhead in the developed implementations is measured. The data proves that Quick-Start is an uncomplicated protocol that could be processed even at line speeds of multiple Gbit/s.

Chapter 7 presents two short applicability case studies: It presents a new concept how fast startup control could be used in order to meet given application performance objectives, such as response time limits. Furthermore, it provides empirical evidence that interactive 3D visualization applications are one promising use case of fast startup congestion control. Such applications are one example of broadband interactive applications. This claim is backed by experiments with a prototypical 3D visualization application that confirm significant performance benefits of fast startup schemes compared to the default TCP congestion control.

Finally, Chapter 8 concludes this thesis and summarizes the major results. It also presents an outlook on potential work items beyond this thesis. Two appendixes complement the document by some additional mathematical background material and a list of configuration parameters that are used in the experiments.

This thesis addresses functional and performance aspects of core Internet protocols, and is thus related to many standardization documents and Internet research efforts. The literature references have been strictly limited to either ground-breaking, fundamental publications, or recent related work. Standard documents, such as *Request for Comments* (RFCs), are only cited if they are of particular importance to this work.

2 Communication network architectures

This chapter introduces the architectures and protocols of packet-switched communication networks. After a review of the fundamental concepts, the Internet as well as IP-based telecommunication network architectures are presented. A specific focus is the realization of the resource management. Furthermore, the expected future evolution of the Internet is addressed.

2.1 Packet networks

2.1.1 Fundamentals and terminology

Modern communication technologies use the principle of *packet switching*. In packet-switched networks, packets are multiplexed in network elements and processed by store and forward mechanisms. A network consists of *nodes*, *links*, and *paths*. Nodes can be defined as network components where the input and output links can have different characteristics. A link is a connection between two of these network nodes. A path is defined as a series of links connecting a sequence of nodes.

Protocols define the behavior required by any entity participating in the exchange of information. Communication in packet switched networks can be connection oriented or connectionless. A *connection* can be defined as a logical relationship between two or more endpoints that exchange data, and it is also known as *virtual channel connection*. A connection can be either uni-directional or bi-directional, and either point-to-point or point-to-multipoint. A *flow* denotes a unidirectional sequence of packets, and a *session* is an abstract temporary association between entities. In general, a session can include several connections, and each bi-directional connection results in at least two flows.

Synchronization is needed to keep the state in different nodes consistent. *Signaling* is defined as the exchange of information for control. Signaling may be realized *in-band* or *out-of-band* [140]. In the former case, signaling data is part of the associated data traffic and typically transported in header fields of the data packets. In contrast, out-of-band signaling messages are separated from the associated data traffic. Out-of-band signaling can be *on-path* or *off-path*. In the on-path case (also labeled data-coupled or path-coupled signaling), signaling messages take the same path like data packets. Off-path signaling (also data-decoupled or path-decoupled signaling) refers to signaling messages that are routed through nodes that are not assumed to be on the data path. Out-of-band signaling typically uses different protocols for the exchange of data and control messages. There is a fundamental trade-off between putting control information in the packet and associating more state information with the protocol [54]. The former approach consumes more bandwidth, whereas the latter one requires more memory in network nodes. The state kept in entities can be either *soft state* or *hard state*. Hard state is explicitly created and removed by messages. Soft state is non-permanent and expires unless it is refreshed.

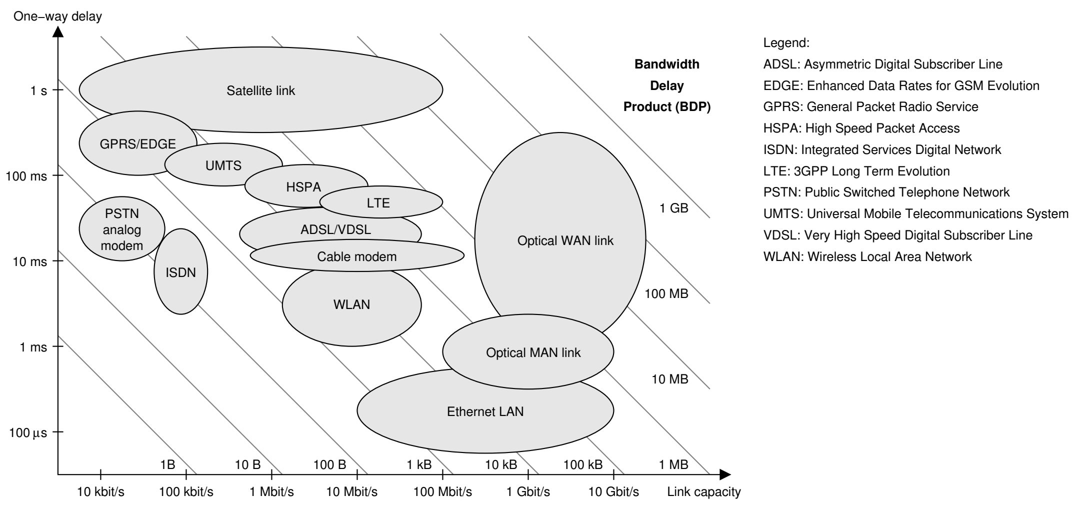


Figure 2.1: Characteristics of selected packet network technologies

One of the most fundamental design patterns in the architecture of packet networks is *layering*. A protocol layer provides an abstraction, i. e., it hides the complexity of the layer below, and it provides a service to the layer above. The mapping of protocol functions to layers is defined by the *network architecture*, which is a set of high-level design principles that guides the technical design of the network, especially the engineering of its protocols and algorithms.

2.1.2 Packet network technologies

A large variety of network technologies can transport packets in *Local Area Networks* (LANs), *Metropolitan Area Networks* (MANs), and *Wide Area Networks* (WANs). Figure 2.1 gives an overview of the characteristics of today's important packet transport technologies, both in fixed and in wireless networks. Circuit-switched time-division multiplex technologies have been omitted if they are not designed for packet transport.

There are several major broadband packet transport technologies: Ethernet is the dominating LAN technology, and its frame format is increasingly also used in MANs and WANs. Broadband fixed access networks mostly use *Asymmetric Digital Subscriber Line* (ADSL) or cable modem technology. Due to a significant technological progress of radio technology there are also more and more wireless networks that allow high-speed packet transfers.

In this thesis, the most important characteristics of packet network technologies are the capacity that they offer, their delay, and potential fluctuations of both metrics. The order of magnitude of the capacity and the one-way delay is illustrated in Figure 2.1. Many access network technologies have an inherent minimum latency that results for instance from transmission encoding. Third-generation cellular networks have a minimum one-way delay of the order of 100 ms. In optical WANs, the one-way delay mainly depends on the path length and is about 1 ms per 200 km fiber length. In state-of-the-art network components, the processing delays can often be neglected [16]. For a given path, an important further metric is the *Bandwidth-Delay Product* (BDP), which is the available bandwidth multiplied by the Round-Trip Time. The RTT is the delay between the sending of a packet and the reception of a corresponding response. It consists of the sum of the two one-way packet delays and the processing times in both endsystems. Figure 2.1 also shows the order of magnitude of the BDP. For instance, a path with an available bandwidth of 1 Gbit/s and an RTT of 200 ms has a BDP of about 25 MB.

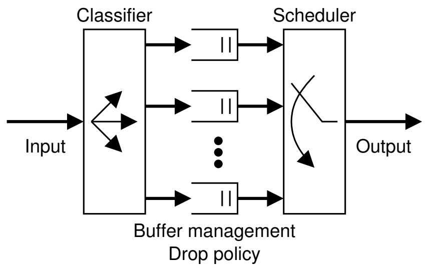


Figure 2.2: QoS differentiation functions

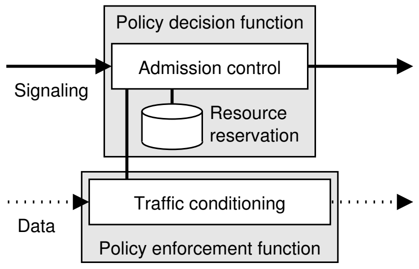


Figure 2.3: QoS assurance functions

2.1.3 Performance and Quality of Service

Packet networks use the principle of statistical multiplexing. As a consequence, the service provided by a packet network depends on many factors, and it may, or may not, meet the expectations of the users. Specific mechanisms can be used to improve the *Quality of Service* of a network. *Quality of Service* (QoS) can be defined as the "collective effect of service performance which determines the degree of satisfaction of a user of the service" [E.800]. This definition highlights operational aspects and is rather vague. This thesis focuses on the quantitative performance aspects of communications QoS [G.1000] as a quality measure. The term *Quality of Service* is therefore used as a synonym for the level of QoS offered by the service provider [G.1000]. *Quality of Service mechanisms* refer to control functions that provide resource assurance and/or service differentiation as shown in Figure 2.2 and Figure 2.3.

A *Service Level Agreement* (SLA) is a contract between a customer and a provider of a service. It contains both technical and non-technical service level specifications. An SLA defines the service, performance metrics, acceptable and unacceptable service levels, liabilities, as well as actions to be taken in specific circumstances, e. g., the compensation in case of SLA violation. A network SLA may specify QoS objectives and then requires deterministic or statistical guarantees. In the latter case, guarantees are allowed to be violated in certain cases.

The simplest service level is *best effort*. It means that the network does its best to deliver data as efficiently as possible, but it does not offer any guarantees. If there are enough resources, a best effort service can satisfy all customers. The resource provisioning scheme that dimensions the network according to the expected demand is called *rightsizing* or *overprovisioning*. However, dimensioning alone cannot avoid that the demand temporarily exceeds the available resources and that *congestion* occurs. Formal definitions of congestion are provided in Section 4.1.1.1.

Two different traffic management solutions deal with scarce network resources: *Traffic control* and *congestion control*. Traffic control refers to all network actions aiming to meet the negotiated performance objectives and to allow the avoidance of congested situations, whereas congestion control refers to all network actions to minimize the intensity, spread and duration of congestion [Y.1221]. Both approaches are mostly orthogonal, since they can be used independently, but there are interactions and similarities. The following sections introduce possible network architectures for traffic control. Congestion control is comprehensively addressed in Section 4, and the specific differences to traffic control are analyzed in Section 4.1.5.2.

2.1.4 Network Quality of Service mechanisms

QoS provisioning in packet networks requires a set of generic network mechanisms [Y.1291]. In the following, these fundamental architectural requirements are briefly introduced. For sim-

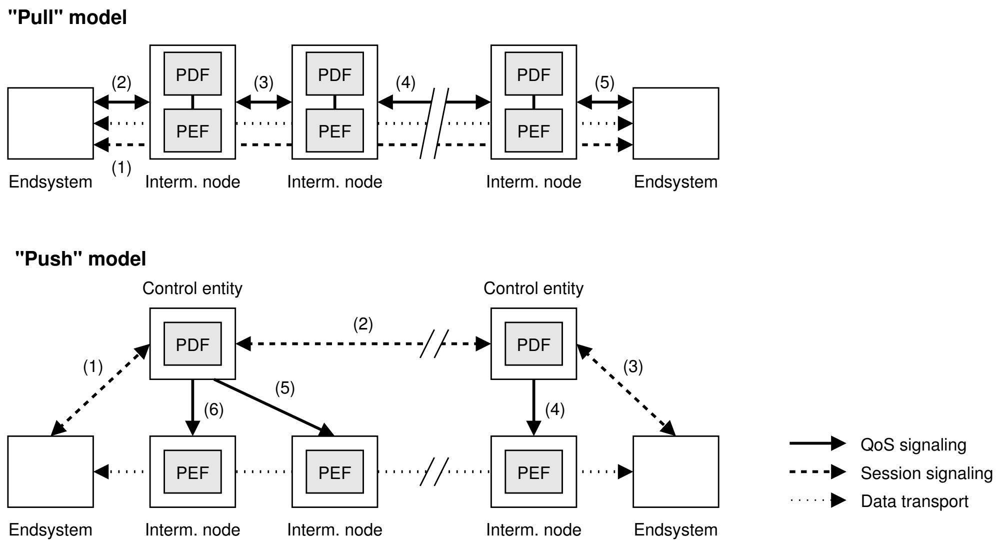


Figure 2.4: Illustration of different QoS signaling architectures and their interaction with session signaling. The typical steps of establishing a session are numbered.

plicity, this section only discusses unicast flows. The same principles can also be applied to connections, sessions, or whole traffic aggregates. Corresponding surveys can be found in references [62, 140, 24, 138], as well as in standard literature on packet networks (e. g., [95]).

The fundamental basis is the ability to differentiate between different QoS requirements (*service differentiation*). The differentiation in network nodes requires mechanisms for *classification*, *marking*, *queue management*, and *scheduling*. The principle node architecture is shown in Figure 2.2. A classifier is an entity that selects and potentially also marks packets according to defined rules. The queue management deals with packets that await processing. There are different schemes that mainly differ in the criteria for dropping packets if the queue fills. The most common strategy is *drop tail* (see also Section 4.2.2.2). A scheduler allocates the capacity of a shared resource among multiple competing users. There are many different scheduling disciplines: The *First In First Out* (FIFO) strategy, which is also known as *First Come First Served* (FCFS), simply serves packets in the order of their arrival. More advanced scheduling algorithms include Static Priority, (Weighted) Round Robin, and Weighted Fair Queueing. The concept of service differentiation can be applied to individual flows, or to aggregates. In the latter case, a *service class* represents a subset of the traffic that requires specific delay, loss, and/or jitter characteristics from the network.

Service differentiation only offers *relative guarantees*, but no *absolute guarantees*, i. e., an isolation of different traffic types. Absolute guarantees require four further mechanisms that are depicted in Figure 2.3: *Traffic conditioning*, *admission control*, *resource reservation*, and *signaling*. Traffic conditioning ensures that a flow adheres to its traffic profile and defined policies. This *Policy Enforcement Function* (PEF) may include metering, policing, shaping, and packet marking. When a new flow is set up, the admission control decides whether there are enough resources available for the new flow, and if it complies to given polices. The admission control is part of the *Policy Decision Function* (PDF). If the flow is admitted, the resource reservation mechanism sets aside the required network resources.

Furthermore, a signaling mechanism is needed to establish, modify, and tear down reservations. A reservation request has to provide a description of the traffic parameters and the desired service parameters (*traffic contract*). Figure 2.4 shows that there are different alternatives for realizing the QoS signaling: In the *pull model* an endsystem requests the required network resources from the network. The request travels through the network node by node. In contrast, in the *push model*, additional entities in the network manage the resources and issue QoS requests. In combination with session signaling the push model requires less signaling because of simpler authorization mechanisms [117].

The provisioning of QoS requires *traffic engineering*, i. e., dimensioning resources, configuring components, and optimizing routing functions so that performance requirements can be fulfilled. Resource dimensioning forms part of the long-term network planning and capacity planing process. Further *Operations, Administration, and Management* (OAM) procedures are required to negotiate and monitor SLAs.

2.2 Internet

2.2.1 Historical evolution

Many aspects of the Internet are coined by its historical evolution. The Internet developed out of research efforts on packet networks that were performed by universities and the Defense Research Projects Agency in the United States in the 1960s and 1970s [129]. By the mid-1970s, early versions of the Transmission Control Protocol were developed for the "ARPANET" by Cerf and Kahn [43]. The base TCP/IP protocol suite was deployed in 1983 and has remained unchanged in many aspects since then. The Internet emerged out of the interconnection of more and more packet networks. In the early 1990s, the World Wide Web (WWW) began to take off, and the Internet became a global information infrastructure interconnecting many hundred millions of endsystems and users.

In 1986, the Internet encountered a major crisis, the so-called "congestion collapse" [RFC 896]. Congested links resulted in long delays that caused timeouts and retransmissions, which made the problem worse. By that time, TCP had flow control mechanisms only. There where discussions to realize congestion control by feedback from the congested network components [54]. However, this would have meant to give up the connectionless model to some extent. In 1988, Jacobson proposed a congestion control scheme to be inserted into TCP [98]. His solution was easily deployable because it required changes in the TCP implementation only. As discussed in Section 4.2, the principles of Jacobson's congestion control are still in use today.

2.2.2 Design principles

The TCP/IP protocol suite has evolved over a period of 30 years through repeated design phases and implementation and testing efforts. It still evolves. Internet standards are developed by the *Internet Engineering Task Force* (IETF) and are published as *Request For Comment* (RFC).

From a retrospective view there have been several goals and design principles of the Internet architecture and its protocols (adapted from [50]):

- Survivability/robustness, i. e., communication continues despite loss of networks or nodes
- Service generality, i. e., support of multiple types of communication services
- Accommodation of a variety of heterogeneous networks
- Distributed management of resources without central control

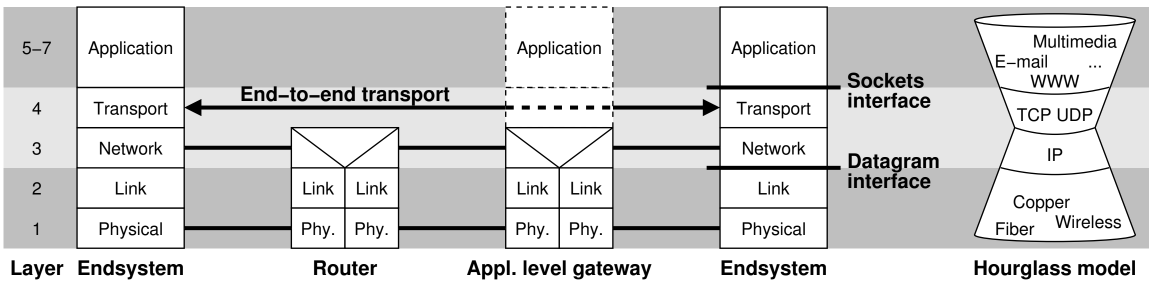


Figure 2.5: Illustration of the Internet protocol stack

The Internet has been designed as a connectionless packet switched network that interconnects collaborating networks and administrative domains. Key design guidelines of the TCP/IP protocol suite are *self-describing datagrams*, *layering*, and the *end-to-end principle* of system design. The end-to-end principle (or end-to-end argument) states that functions shall not be performed by the communication system if they can “completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system” [188]. It is closely related to the *fate-sharing principle*, which mandates that “the intermediate packet switching nodes, or gateways, must not have any essential state information about on-going connections” [50]. Furthermore, “end-to-end protocol design should not rely on the maintenance of state (i. e., information about the state of the end-to-end communication) inside the network” [RFC 1958]. The Internet philosophy is to provide services at the endsystems, whereas the network only realizes packet transport [RFC 1958].

2.2.3 Network architecture

The Internet protocol stack distinguishes five layers that are depicted in Figure 2.5:

1. The *physical layer* includes mechanisms required to transmit bits over a physical medium.
2. The *link layer* aggregates bits to frames and transmits the frames between network nodes. Link layer protocols may also provide error correction and flow control.
3. The *network layer* (also called *Internet layer* or *Internet Protocol layer* or *internetwork layer*) forwards packets through a network of links.
4. The *transport layer* manages the end-to-end transport of data.
5. The *application layer* consist of protocols for application-specific communication.

The *Open Systems Interconnection* (OSI) reference model [ISO 7498] developed by the *International Organization for Standardization* (ISO) uses in total seven layers. The application layer in the Internet stack corresponds to the upper three OSI layers. However, these three layers can hardly be mapped directly to the existing application protocol functionalities [54].

The Internet stack is often illustrated by the *hourglass model* as depicted in Figure 2.5, since the network layer is a convergence layer over many heterogeneous link layer technologies, with a large variety of applications being realized on top of it. End-to-end transport between application instances is realized at the transport layer. In this thesis, the endpoints of a connection are named *endsystems* in order to avoid the ambiguous term *hosts*. Interworking nodes that process IP packets are labeled *routers*. *Application Level Gateways* (ALGs) transparently process transport and/or application protocols. The term *packet* is used for network layer data units, whereas *segment* refers to transport layer data units.

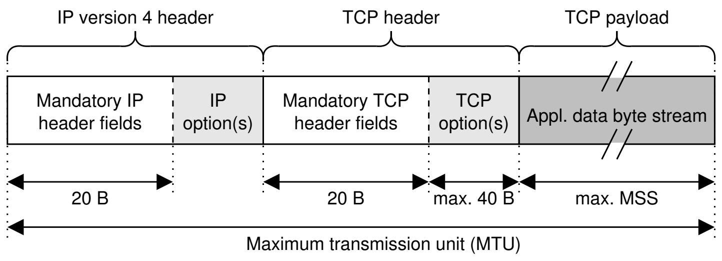


Figure 2.6: Structure of a TCP segment encapsulated in an IP packet, including length indications. The shaded fields are optional.

The network layer requires only a simple interface to the link layer and just assumes that datagrams can be transported. Network and transport layer are closely linked. The *de facto* standard interface between transport and application layer is the *sockets interface* [1003.1]. This interface originates from Unix operating systems and has later been adopted by other operating systems as well. A socket is a special type of a file handle. There are two main transport services: Unreliable datagram and reliable byte stream-oriented transport. The sockets interface also separates the *kernel space* from the *user space* in many operating systems.

2.2.4 Protocols

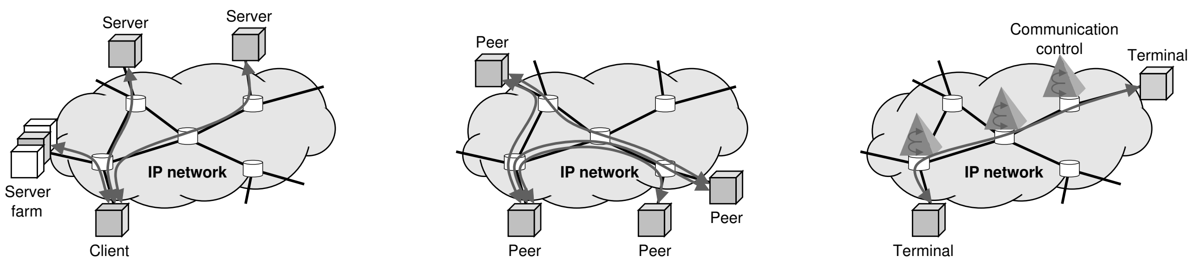
In the following, the Internet protocols are briefly introduced as far as they are relevant for this thesis. Further details can be found in computer network literature (e. g., [95]).

In the network layer, the Internet Protocol (IP) [RFC 791] layer offers an unreliable, connectionless delivery service for variable-size packets. Routers forward IP packets according to *routing tables*, which are either statically configured or dynamically learned from *routing protocols*. In the Internet, inter-domain routing between *Internet Service Providers* (ISPs) is controlled by the *Border Gateway Protocol* (BGP). The network interfaces are identified by IP addresses, which are 32-bit numbers in IP version 4.

Traditionally, there have been two transport layer protocols in the Internet: The Transmission Control Protocol (TCP) and the *User Datagram Protocol* (UDP). UDP offers connectionless, unreliable transport of datagrams and is basically a multiplexing layer on top of IP.

The Transmission Control Protocol (TCP) is a connection-oriented, bidirectional, point-to-point transport protocol with reliable, in-order data delivery. TCP transports a serial byte stream (a "byte-pipe") between applications and manages the recovery from erroneous, lost, or duplicate segments. In the source, the byte stream is fragmented into appropriately sized segments with a *Maximum Segment Size* (MSS) according to the *Maximum Transmission Unit* (MTU) of the path. The resulting packets are passed to the IP layer and reassembled at the destination. Figure 2.6 sketches the resulting structure of an application byte stream that is transported in a TCP segment and encapsulated in an IP packet. The TCP header, as well as the IP header, can be extended by header options. TCP options are frequently used, in particular during the initial state synchronization by the *three-way handshake*. TCP is a window based protocol that realizes both *flow control* and *congestion control*. These functions are detailed in Section 4.2.

TCP was originally standardized in [RFC 793]. As TCP has evolved over the years, many other documents have become part of the accepted standard for TCP. A large number of more experimental modifications to TCP have also been published in the RFC series, along with



(a) Client-server paradigm (b) Peer-to-peer paradigm (c) Telecommunication paradigm

Figure 2.7: Comparison of different services delivery principles

informational notes, case studies, and other advice [RFC 4614]. As a consequence, there has never been a unique standardized protocol description. Instead, details have been left to the implementers, and this trend continues as TCP evolves.

Another protocol on top of IP is the *Internet Control Message Protocol* (ICMP). Two further transport protocols are the *Stream Control Transmission Protocol* (SCTP) and the *Datagram Congestion Control Protocol* (DCCP). SCTP has been developed for signaling applications, but it is a general purpose protocol. It provides a reliable datagram service with an additional multiplexing layer and it supports multi-homing. The error recovery, flow control and congestion control mechanisms are similar to those of TCP. The Datagram Congestion Control Protocol (DCCP) provides bidirectional, unicast, unreliable, congestion-controlled datagram service and is intended for streaming media applications. In DCCP, an application has a choice of congestion control mechanisms, each specified by a *Congestion Control Identifier* (CCID). CCID 2 defines a TCP-like congestion control, whereas CCID 3 is a rate-based TCP friendly congestion control mechanism. Until now, neither SCTP nor DCCP are widely used in the Internet.

In the application layer, a large variety of protocols exists. One core protocol is the *Hypertext Transfer Protocol* (HTTP) [RFC 2616]. Furthermore, the *Domain Name System* (DNS) provides address mapping from *Uniform Resource Identifier* (URI) in text format to numerical IP addresses. Their usage in interactive applications is presented in Section 3.1.2.

2.2.5 Services

The Internet offers a large set of services including the WWW, e-mail, file transfer, gaming, etc. Websites also integrate more and more multimedia content such as audio and video clips. The combination of the TCP and HTTP is the *de facto* standard in the WWW. As illustrated in Figure 2.7(a), the WWW is a global client-server system. The clients are Web browsers that communicate with Web servers over the Internet. An alternative to client-server applications is the *Peer-to-Peer* (P2P) paradigm that uses no or only few centralized infrastructure (cf. Figure 2.7(b)). Peer-to-peer applications are widely used for file sharing, and also in *Voice-over-IP* (VoIP) and *Internet Television* platforms. The vast majority of the Internet traffic results from the bulk data transport of peer-to-peer file sharing platforms.

2.2.6 Network Quality of Service mechanisms

The TCP/IP protocol suite, originating from academic and military networks, does not incorporate Quality of Service mechanisms. The Internet traditionally transports traffic on a best effort

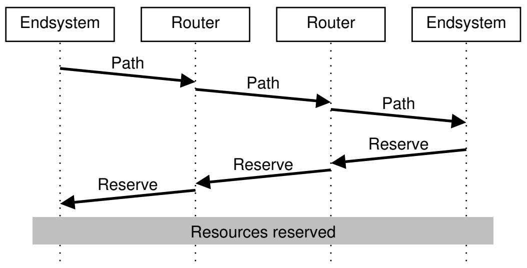


Figure 2.8: Receiver-initiated QoS reservation (e. g., RSVP)

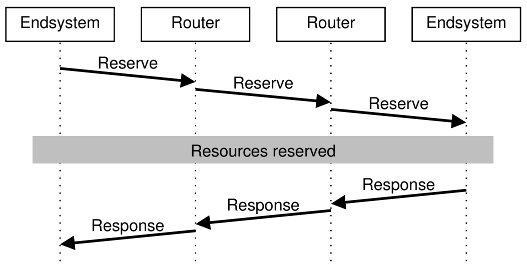


Figure 2.9: Sender-initiated QoS reservation (e. g., NSIS sender-initiated mode)

basis in combination with endsystem-based congestion control. The underlying assumption is that all applications are elastic [212]. As the Internet is becoming a critical infrastructure for the society, it has been observed that a certain subset of the traffic may need performance guarantees [RFC 1287]. The IETF has developed and standardized several QoS architectures that are briefly summarized below. Long surveys can be found in the references [62, 24, 138].

The *Integrated Services* (IntServ) [RFC 1633] architecture has been developed in order to provide absolute QoS guarantees to individual flows. IntServ uses resource reservation according to the *pull model* and three classes of service: *guaranteed service*, *controlled load service*, and *best effort service*. The guaranteed service assures a minimum bandwidth and upper bounds on the delay. Controlled load refers to a service that is equivalent to best effort in a lightly loaded network. Admission control and traffic policing is used to achieve low delays and avoid packet losses, but there are no quantitative assurances, and resources may be oversubscribed.

In the IntServ architecture, routers on the path of a flow maintain soft state about the allocated resources. The corresponding signaling protocol is *Resource Reservation Protocol* (RSVP). RSVP is an on-path, receiver-initiated protocol for the setup of unidirectional resource reservations for unicast or multicast flows. As shown in Figure 2.8, RSVP uses a two-way message exchange with two main messages types: The *path* messages originates from the flow source. It is routed towards the destination by the IP routing protocols and includes a router alert IP option. Each RSVP router processes the message and creates routing path state. The *reserve* messages are sent back along the path and request for resources. The routers analyze the included *flow specification* and either accept or reject the request. In order to update the soft state, the messages are repeated periodically.

RSVP suffers from several limitations such as the overhead due to the multicast-driven design and the lack of security mechanisms [RFC 4094]. This is why the IETF *Next Steps in Signaling* (NSIS) working group develops a successor protocol [74]. The objective is a general purpose, extensible, on-path signaling protocol suite for control of network entities. The modular design consists of a transport layer and a signaling layer. A signaling protocol has been standardized for QoS resource reservation. It enables both sender and receiver initiated reservations (see Figure 2.9) and is more flexible than RSVP. There are also protocols for the control of middleboxes such as firewalls or entities performing *Network Address Translation* (NAT) [117].

Some inherent problems of the IntServ architecture cannot be solved by improved protocols: Keeping per-flow state in network nodes results in a significant processing effort and does not scale to a large number of flows. The usage of the router alert option in the IP header also raises concerns since it affects the router performance. Furthermore, IntServ only specifies the resource management. Separate protocols and mechanisms are required for *Authentication,*

Table 2.1: Comparison of the service classes defined by IETF, IEEE, and ITU-T

Service class according to			**Application**	**Tolerance to ...**		
IETF [RFC 4594]	**IEEE** [802.1D]	**ITU-T** [Y.1541]	**example**	**Loss**	**Delay**	**Jitter**
Network control	Network control	—	Network routing	Low	Low	Yes
Telephony	Voice	Class 0	IP telephony bearer, circuit emulation	Very low	Very low	Very low
Signaling		Class 2	IP telephony signaling	Low	Low	Yes
Real-time interactive		Class 0	Video conferencing, interactive gaming	Low	Very low	Low
Multimedia conferencing		Class 0	(Adaptive) video conferencing	Low – medium	Very low	Low
Broadcast video	Video	Class 1	Broadcast television and live events	Very low	Medium	Low
Multimedia streaming	Controlled load (equiv.)	Class 4	Streaming video and audio on demand	Low – medium	Medium	Yes
Low-latency data	Excellent effort	Class 3	Client/server trans., e-commerce	Low	Low – medium	Yes
OAM		Class 3	OAM for configuration and management	Low	Medium	Yes
High-throughput data		Class 4	Store and forward applications	Low	Medium – high	Yes
Standard	Best effort	Class 5	Undifferentiated applications	Not specified		
Low-priority data	Background	—	Flows that need no bandwidth assurance	High	High	Yes

Authorization, Accounting (AAA) and other signaling tasks. Applications have to know *a priory* their traffic profile and have to wait until the reservation is complete. This may delay the data delivery and becomes problematic for short-lived sessions.

The *Differentiated Services* Framework (DiffServ) [RFC 2475] has been developed to overcome the scalability problems and operational issues of IntServ. DiffServ provides service differentiation between several traffic classes, which receive a well-defined *Per Hop Behavior* (PHB) with different priorities at each DiffServ router. Packets are classified according to the *DiffServ Code Point* (DSCP) in the IP header. The DiffServ classification, aggregation, and traffic conditioning is typically realized by edge routers, while the priority mechanisms must also be implemented by the core routers within a domain. In addition to the basic *default forwarding* for best effort, DiffServ routers provide two types of PHB: *expedited forwarding* and *assured forwarding*. The former is the highest DiffServ priority class and emulates a *virtual leased line*. The latter defines twelve classes with different queue management schemes and scheduling priorities. Table 2.1 lists the DiffServ service classes and compares them to service classes that have been defined by the *Institute of Electrical and Electronics Engineers* (IEEE) and the *International Telecommunications Union* (ITU). As DiffServ deals with aggregates rather than single flows, no per-flow state and no out-of-band signaling is required. Therefore, the architecture scales better than IntServ, at the cost of a less stringent QoS assurance.

In the current Internet, DiffServ is deployed within certain domains, but not globally supported. In many domains, some service differentiation is realized below IP: Since many link layer technologies distinguish between a number of traffic classes, the Diffserv classes can be mapped to these service levels (cf. Table 2.1). There are also attempts to implicitly classify flows by *Deep Packet Inspection* (DPI). *Multiprotocol Label Switching* (MPLS) is widely used, too. It offers a virtual circuit model below IP and enables service differentiation and guarantees. The utilization of these techniques across different administrative domains is an unsolved issue.

2.3 IP-based telecommunication networks

2.3.1 Historical evolution

Telecommunication networks such as the *Public Switched Telephone Network* (PSTN) are much older than packet networks. The PSTN is a circuit-switched network that guarantees a high service quality. However, compared to circuit switching, packet switching proved to be a cheaper and more flexible technology for network interconnection. This is why telecommunication services are more and more realized with the available IP technology. Still, the open design philosophy of the Internet differs significantly from the classical telecommunication network architecture with managed services, dumb terminals, and an intelligent network [54]. This also implies that the existing telecommunication business models and revenue streams of the network operators do not work well in the Internet with its different network interconnection arrangements [144]. To cope with this trend, telecommunication standardization bodies develop *Next Generation Networks* (NGNs) as a new, alternative architecture for IP networks.

2.3.2 Design principles

The term Next Generation Network can be defined as a “packet-based network able to provide telecommunication services and able to make use of multiple broadband, QoS-enabled transport technologies and in which service-related functions are independent from underlying transport-related technologies” [Y.2001]. The NGN involves various standardization bodies, including the *International Telecommunications Union* (ITU), the *3rd Generation Partnership Project* (3GPP), the *European Telecommunications Standards Institute* (ETSI), as well as standardization bodies for ADSL and cable networks, and other national organizations. Since the NGN uses IP technology, many IETF protocols are adapted. The objective is to build managed IP networks that can provide QoS guarantees and security through resource management and user authentication. These functions are provided by stateful application layer entities in the network (see Figure 2.7(c)). Even if the NGN interfaces use standardized protocols, the architecture results in closed networks, which are also named “walled gardens”.

2.3.3 Network architecture

NGN architectures separate services from transport [Y.2011] and can be subdivided into a *transport stratum*, which provides connectivity and data transfer functions, and a *service stratum*, which includes functions that control user services. It is also common to introduce a third *application stratum* for application-level *Service Delivery Platforms* [174], which are not entirely standardized. Each stratum comprises one or more layers, where each layer is conceptually composed of a *data plane* (or *user plane*), a *control plane*, and a *management plane*.

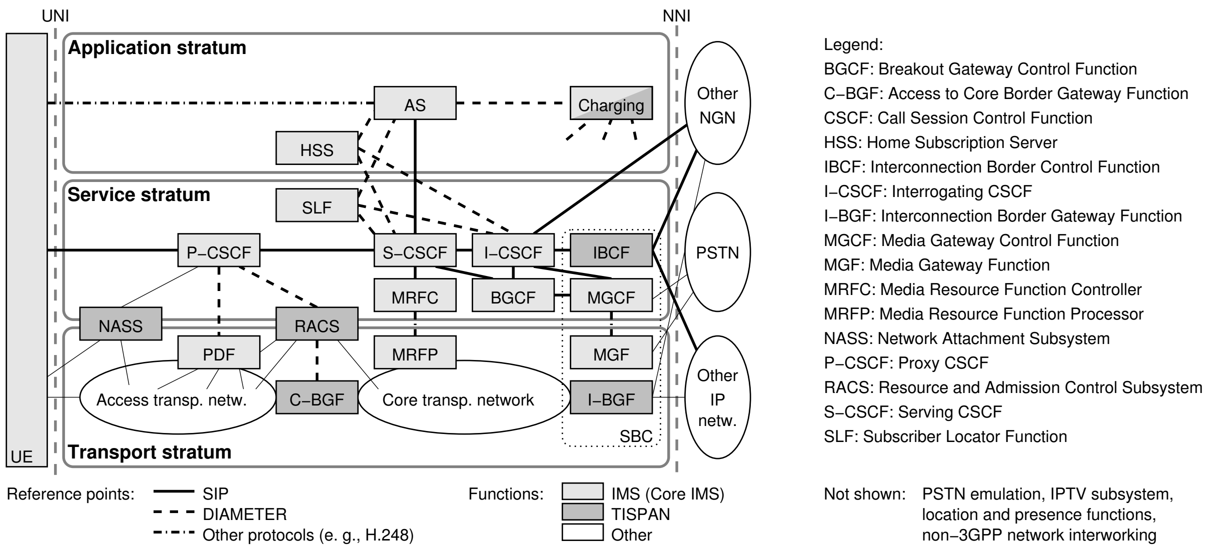

UNI
NNI
Application stratum
Service stratum
Transport stratum
AS
Charging
HSS
SLF
P–CSCF
S–CSCF
I–CSCF
IBCF
MRFC
BGCF
MGCF
NASS
RACS
PDF
MRFP
MGF
C–BGF
I–BGF
SBC
Access transp. netw.
Core transp. network
UE
Other NGN
PSTN
Other IP netw.
Legend:
BGCF: Breakout Gateway Control Function
C–BGF: Access to Core Border Gateway Function
CSCF: Call Session Control Function
HSS: Home Subscription Server
IBCF: Interconnection Border Control Function
I–CSCF: Interrogating CSCF
I–BGF: Interconnection Border Gateway Function
MGCF: Media Gateway Control Function
MGF: Media Gateway Function
MRFC: Media Resource Function Controller
MRFP: Media Resource Function Processor
NASS: Network Attachment Subsystem
P–CSCF: Proxy CSCF
RACS: Resource and Admission Control Subsystem
S–CSCF: Serving CSCF
SLF: Subscriber Locator Function
Reference points: SIP
DIAMETER
Other protocols (e. g., H.248)
Functions: IMS (Core IMS)
TISPAN
Other
Not shown: PSTN emulation, IPTV subsystem, location and presence functions, non–3GPP network interworking

Figure 2.10: Overview of the architecture of 3GPP IMS and ETSI TISPAN. In order to simplify the picture, several details have been omitted.

The core of the current NGN standards is the *IP Multimedia Subsystem* (IMS), which has originally been developed by 3GPP for cellular networks [TS 23.228]. The IMS is an access-independent, IP-based service control architecture that enables various types of mobile multimedia services, such as audio streaming, presence, messaging, push-to-talk, and conferences. The original IMS was later adopted by other standardization bodies, in particular ETSI and ITU-T. ETSI's *Telecommunications and Internet Converged Services and Protocols for Advanced Networks* (TISPAN) [ES 282 001] extends the IMS for use of non-3GPP and fixed access networks. The IMS is also the core of the ITU-T NGN architecture [Y.2011]. The different standards differ in some details and also use a slightly different terminology.

The IMS standardizes functional blocks and interfaces for session control, subscriber data management, interworking, and charging. Figure 2.10 provides a simplified overview of the IMS architecture and its TISPAN extensions. The core of the architecture are different types of *Call Session Control Functions* (CSCFs). They handle the registration of the *User Equipment* (UE), i. e., the endsystem, and the session signaling. The *Home Subscription Server* (HSS) is the main database and contains user identities, profiles, and authentication and authorization information. Value-added services are executed by *Application Servers* (ASs). IMS also defines reference points for offline and online charging. There can also be further service-related functions for multimedia resources and interworking with other IP networks or the PSTN. Several books give a complete overview of the functions and reference points [40, 178].

TISPAN extends the IMS by two subsystems: The *Network Attachment Subsystem* includes functions to identify and authorize customers and to configure the subscriber line. The *Resource and Admission Control Subsystem* is responsible for policy control, resource reservation, and admission control. Another extension are border elements for network interconnection.

2.3.4 Protocols

The IMS uses the *Session Initiation Protocol* (SIP) for session signaling at internal reference points, at the *User Network Interface* (UNI), and at the *Network Network Interface* (NNI). IMS also utilizes further IETF protocols, such as the *Session Description Protocol* (SDP) or the *DIAMETER* protocol (cf. [40, 178]). In IETF terminology, the CSCF and AS are basically combinations of *SIP user agents*, *SIP proxies* and *SIP back-to-back user agents*. The border gateway functions correspond to entities that are known as *Session Border Controllers* (SBCs), which provide a variety of functions to enable or enhance session-based multimedia services, including access control, topology hiding, *Denial-of-Service* (DoS) detection and prevention, NAT traversal, application-layer protocol interworking, and traffic management [85, 117].

2.3.5 Services

The main driver for the NGN development is *convergence*, i. e., the integration of telecommunication, multimedia and other data services in a unified service control platform. The vision is that network operators offer service delivery platforms that support a wide range of services and interfaces to the *operational support systems* and *business support systems*, as well as standardized interfaces for third-party providers. Important components are content portals and provisioning systems, access control and security functions, customer relationship management and subscriber data storage, and pre-paid/post-paid billing systems. With such platforms, network operators could play a central role in service provisioning and offer more than simple “bit pipes”. An example is the converged provisioning of telephony services, *Internet Protocol Television* (IPTV), and Internet access (“triple play”). Other examples are new platforms for music and gaming, which are among the top requested mobile applications [174].

With few exceptions, the NGN standards of 3GPP, ETSI, and ITU-T do not define these services, but only *service enablers*, such as access to presence or location information, single-sign-on mechanisms, AAA, or QoS mechanisms. Actual service definitions exist mainly for linear multimedia services, in particular for IP-based telephony and IPTV. Concerning telephony, the objective of TISPAN is to realize *PSTN simulation*, i. e., PSTN-like services for IP telephones, but no backward compatibility (*PSTN emulation*). There are also activities to realize IPTV over IMS, as well as efforts to realize Web-like applications within the IMS (e. g., [217]).

2.3.6 Network Quality of Service mechanisms

The ability to offer predictable end-to-end service quality is an important design objective of NGN platforms. The original 3GPP IMS only covers QoS guarantees in the access networks. For 3GPP cellular networks, four different QoS classes are defined [TS 23.107]: The *conversational*, *streaming*, *interactive*, and *background* class have different link layer attributes, including the maximum rate and a guaranteed rate. The IMS offers interfaces to the corresponding link layer resource management.

In contrast, ETSI TISPAN has the objective to provide end-to-end QoS guarantees. TISPAN’s *Resource and Admission Control Subsystem* standardizes QoS policy decision and policy enforcement functions that can realize traffic control. Different to the IETF QoS architectures, TISPAN focuses on the *push model*. Figure 2.11 illustrates the integration of the QoS and session signaling, assuming a multimedia application. In order to establish a session, the UE sends a SIP message to the responsible CSCF. After capabilities and media parameters have been negotiated, the IMS can reserve suitable resources. However, Figure 2.11 also reveals that a significant number of signaling messages is required before a session is established. A timely session setup requires therefore a very fast processing of signaling messages [238].

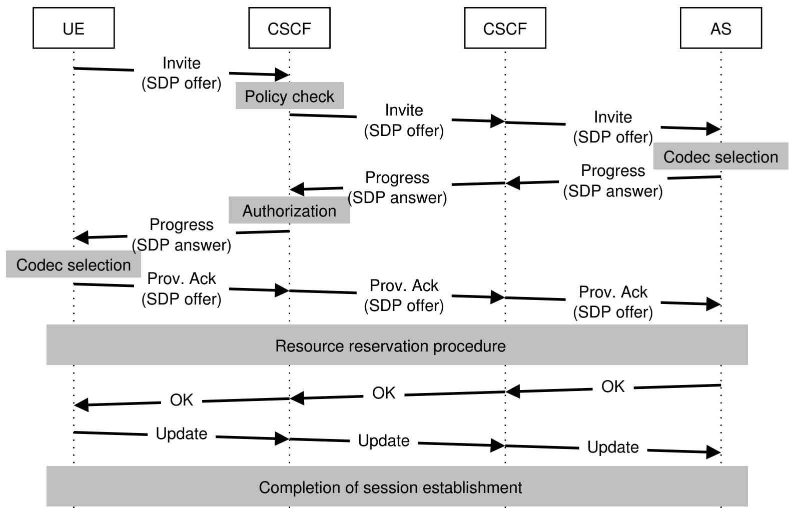


Figure 2.11: Example of a SIP session establishment in the IMS (adapted from [217]). Additional signaling handshakes may be required within the shaded fields.

2.4 Future evolution

2.4.1 Technological progress and trends

The world has changed since the original design of the Internet, driven by several technological and socio-economic trends [142]: First, due to the technical progress in transmission and router technology, the bandwidth of Internet links has increased by many orders of magnitude, and it continues to grow rapidly. Broadband Internet connectivity is more and more the norm. At the same time, the Internet traffic has increased exponentially as well. New information-centric application areas emerge, e. g., in the health sector, and new bandwidth-demanding applications continue to create an insatiable demand for bandwidth. Second, the *mobile Internet* is becoming ubiquitous, as more and more smart and powerful mobile devices can conveniently access the Internet and WWW. Third, the base of Internet users, their preferences, and their usage patterns have shifted towards personalization of services (*social networks*) and *user generated content* (*Web 2.0*). The World Wide Web is today a large commercial platform that allows online access to more and more of the world's information.

Unlike the telecommunication sector, the Internet world was able to embrace these trends due its open environment, its large community of *Information Technology* (IT) application developers, and its fast, low cost development cycles favored by the end-to-end design. Most commercially deployed service delivery platforms are in fact IT-based systems and not telecommunication service control frameworks [174]. Further reasons for the slow market penetration of NGN platforms are the complexity of the architecture, unsolved inter-domain interoperability issues, the slow standardization process, the lack of applications and suitable client devices, as well as the fundamental architectural contradiction between network-based service control and the decoupling of applications and network that is inherent to the TCP/IP protocol suite.

2.4.2 Architectural challenges for the Future Internet

The Internet was never designed to be a critical part of an economy's infrastructure [142]. In the early 1990ies discussions started about the development towards a *Future Internet* [RFC 1287].

The motivation was the shortage of IP version 4 addresses, which triggered the development of IP version 6. Since then, further problems have been identified [51, 19, 142, 67]:

- *Security*: The Internet suffers from many security vulnerabilities, including unwanted traffic (e. g., e-mail spam, DoS attacks), distribution of viruses, identity thefts, and privacy infringements. The original Internet architecture was primarily designed for a benign and trustworthy environment, with little or no consideration of security issues [19].
- *Addressing and routing*: The routing system faces significant scalability challenges, which are manifested by a rapid increase of routes injected by BGP. The hierarchical aggregation of prefixes is considered to be broken due to provider-independent addressing, site multi-homing, and traffic engineering. There is a semantic overloading of the IP address, which serves as endpoint identifier, endpoint locator, and forwarding identifier, which inherently conflicts with mobility and multi-homing [142]. A related problem is the sockets interface, since applications have to be aware of both IP addresses and DNS names.
- *Dependability and robustness*: Internet services have a typical downtime of several hours per year, which is by far worse than the PSTN availability of about 99.999 %. Furthermore, the robustness is affected by long BGP convergence times [156].
- *Lack of flexibility of the end-to-end protocols*: It is questioned whether the design of the TCP/IP suite and in particular TCP as default Internet transport protocol is indeed future-proof [12]. This question is addressed more in detail in Chapter 4. Furthermore, the Internet end-to-end design and layering is increasingly violated by middleboxes (NATs, SBCs, . . .) and intermediate layers [51]. The required interworking increase operational complexity and cause many problems for applications.

Additional research challenges are network management (monitoring and diagnosability) [19, 67], the support for autonomic operation [142], better support for mobile applications [142, 67], other semantics and alternatives to strict protocol layering [51], and accountability [51, 142]. Making the Internet more viable and profitable is also mentioned as a design goal [67]. There are also arguments in favor of circuit switching in particular in optical core networks [156].

The lack of Quality of Service support could be seen as a key problem of today's Internet. However, the opinions vastly diverge [RFC 5290], and there are several non-architectural reasons for the lack of QoS provisioning in today's Internet:[1]

- *Insufficient benefits*: Even multimedia applications such as video streaming do not inherently require QoS mechanisms. As argued for instance in references [167, 144], only few conversational services such as voice and video telephony actually require data delivery with hard deadlines. These applications create only a small fraction of the total traffic in the Internet, and a weighted congestion control can achieve a prioritization without network support (see Section 4.2.3.2). Other multimedia applications, such as audio and video streaming, can use adaptive codecs and buffering in the receiver [208]. The content can be progressively downloaded with faster-than-real-time transfer, e. g., using HTTP over TCP. Even live television can be delivered by faster-than-real-time transfers when a small delay lag of one or two seconds can be tolerated [167, 144].
- *Standardization issues*: Further standardization and agreement among providers is needed, since common service class definitions and mappings as well as commonly agreed performance measurement approaches and metrics are missing [97]. Also, extensions of the Internet interdomain routing protocols and interfaces would have to be standardized.

[1]Network layer multicast, which is sometimes also listed as a desired Internet feature, faces similar challenges.

- *Deployment challenges*: End-to-end QoS support shares two characteristics with other slow-deploying functions: The incremental deployment properties are poor, since there is only limited benefit until it is widely deployed. And it requires cooperation among a large number of organizations. This results in few incentives to start an initial deployment.
- *Economic aspects*: Service differentiation is a business issue. End users have varying views of which applications should be prioritized, and most customers are not willing to pay for a performance difference that they cannot perceive. In a rightsized network, best effort transport is sufficient most of the time. This makes it difficult to create a business case in the consumer market. In order to be meaningful, QoS guarantees must be authorized and charged and thus cause economic challenges in the network interconnection [144]. Network operators also have no incentive to prioritize certain Internet traffic such as VoIP to ensure a high service quality, given that these Internet applications compete with their comparable NGN services. QoS mechanisms also raise the question of *network neutrality* [167]. A strong opposition against traffic differentiation mechanisms is caused by the fear that differentiation may result in discrimination and censorship.

2.4.3 Research directions

It is commonly agreed that more research is needed to overcome the open issues of security, addressing/routing and dependability in the Internet. Most of the design limitations of the Internet cannot be removed by incremental evolutionary enhancements. This motivated the research community to rethink the fundamental design principles and to explore potentially disruptive alternatives in order to answer the question how a network would look like if the design could start "from scratch" [51, 19]. Even though the design objectives and principles of such a *clean slate* network architecture are unclear, there are many research activities, which are surveyed e. g. by Feldmann [67]. The bottom line of most work is that the most of the Internet design principles (cf. Section 2.2.2) can remain unchanged.

But it is also believed that today's Internet is too inflexible to allow the deployment of new innovations. A major focus of Future Internet research is building experimental research platforms that enable large-scale experiments. They are motivated by significant differences between theoretical paper studies and real experiences [177]. Although theoretical analysis, simulation, or emulation can be useful tools to evaluate a new concept, experiments with real implementations are necessary as well in order to test interaction with unexpected events in a realistic environment. There are several ongoing activities to build the required experimentation platforms, which use programmable network components, virtualization, federation, and slice-based experimentation so that even disruptive technologies can be tested at global scale.

The work presented in the following chapters is inspired by these trends in Future Internet research. The hypothesis is that many further Web applications will emerge, that broadband best effort connectivity will be the norm, and that congestion control instead of NGN technologies will be the cornerstone of the global network resource management. Under these constraints the clean slate thinking requires to question whether the existing Internet congestion control principles are still valid, or whether alternative schemes could be possible. The design of the flow startup is one of the core open problems [12]. The following chapters study the constraints and implications of such alternative mechanisms and thereby contribute to the "Post-TCP" research [198], i. e., the question how end-to-end transport could be realized in future.

3 Performance of broadband interactive applications

Networked applications, in particular in the WWW, are realized by a complex interplay of functions in the endsystems and their communication. The user experience is determined by the application performance (*Quality of Experience*). Network performance alone, as presented in the previous chapter, is not sufficient to achieve a high user satisfaction. This chapter focuses on the application-level performance of interactive applications in the WWW. It starts by a brief review of realization techniques for Web applications and a definition of the term *broadband interactive application.* The next section reviews the state-of-the-art methods to achieve and assure reasonable application performance, which range from mechanisms in endsystems to global distribution mechanisms. These *application QoS techniques* have similarities with network QoS mechanisms, but also fundamental differences. Finally, the chapter discusses methods for the evaluation of the performance and scalability of interactive applications.

3.1 Internet and Web applications

3.1.1 Classification of applications

Networked applications can be classified along various dimensions. The transported data can be continuous linear data streams, e. g., in conversational audio or video communication, or it can consist of discrete downloadable content. Unlike downloadable content, a *data stream* is a continuous sequence of data without predetermined end. Streaming traffic is typical for playback applications. Interactive applications usually use downloadable content that is transported in form of *transactions*. One further difference between these two classes is the required *reliability*: The codecs for media streams can often tolerate some moderate amount of lost packets, while data-oriented applications are error-intolerant and require a reliable transport [G.1010].

Another fundamental criterion for classifying applications is the *elasticity*, i. e., their ability to adapt their communication parameters to the available resources. Even though different classes of applications exist, there is no agreed terminology: Shenker distinguishes between *real-time*, *adaptive*, and *elastic* applications [212]. The former class is characterized by hard delay bounds for data delivery, whereas the latter class is rather tolerant to delays. Adaptive applications are between both extremes, i. e., they are tolerant to delay jitter, but they have intrinsic bandwidth requirements. Another taxonomy of applications [RFC 1633] uses the terms delay *intolerant playback*, *tolerant playback*, and *elastic* applications. [G.1010] divides applications into four different delay tolerance categories, namely, *interactive*, *responsive*, *timely*, and *non-critical*. Finally, reference [62] distinguishes between the dimensions *(non)interactive*, *(in)elastic*, *(in)tolerant*, and *(non)adaptive*.

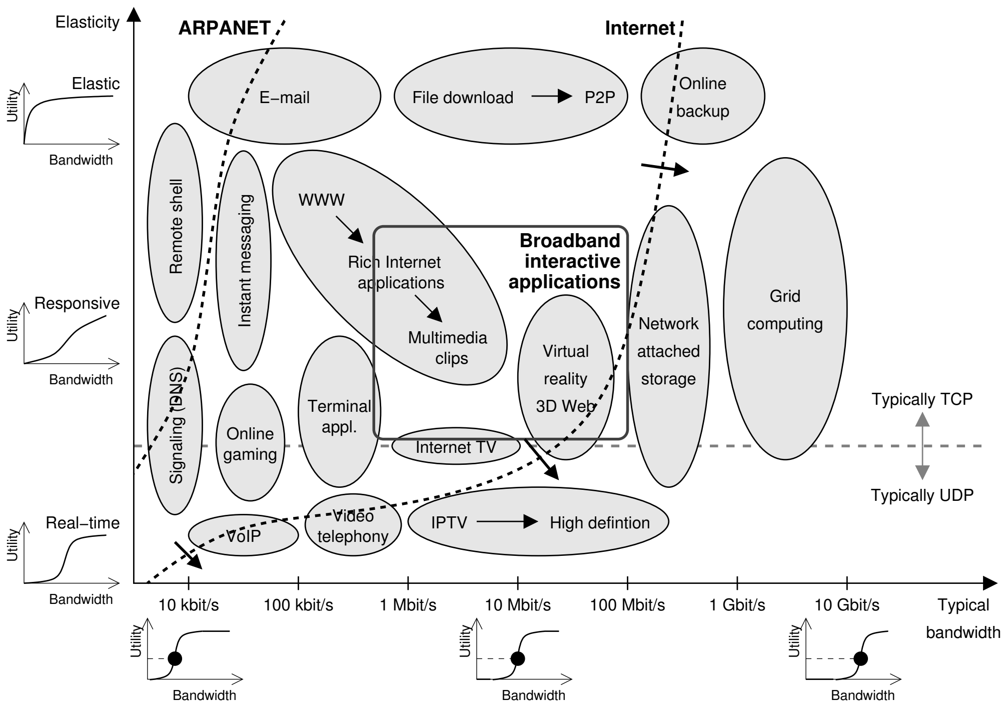
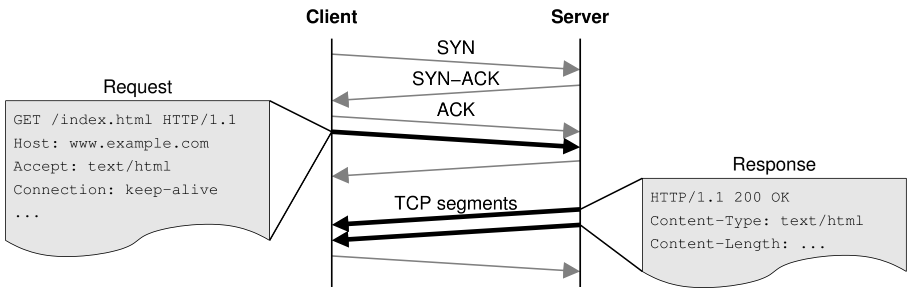


Figure 3.1: Internet application portfolio (originally presented in [198]). The shape and a minimum threshold of the utility function are used as dimensions. Broadband Internet connectivity enables new interactive applications in the center and in the right part of the portfolio.

This thesis uses a modified version of the taxonomy of [212] and distinguishes between *real-time*, *responsive*, and *elastic* applications. The expression *adaptive* is avoided because it also refers to functional aspects and is thus ambiguous: The dynamic change of communication parameters actually realizes a *parametric adaptation* only, whereas an application can also change its functional structure and behavior during run-time (*compositional adaption*) [149]. The proposed classification can be formalized by help of the *utility function* $U_i(x_i)$, which quantifies the utility U_i of an application as a function of the available bandwidth x_i (see also Section 4.1.1.4). Different degrees of elasticity result in a different shape and gradient of that function. Most applications also require a minimum bandwidth to be useful, i. e., to exceed a minimum utility. The combination of the shape and the minimum classifies applications in a two-dimensional portfolio that is depicted in Figure 3.1. This classification is also closely related to the usage of transport protocols; most applications in the upper part of the portfolio use TCP transport.

3.1.2 Web application technologies

Web applications are a special form of client-server applications. The client is typically a *Web browser* that provides the *Graphic User Interface* (GUI), realizes the user interactions, and retrieves data from *Web servers*. The basic interaction is a *request*, which consists of the query of a client and the server's response. In traditional Web applications, a user issues requests for Web pages, which are multipart hypertext documents. In the simplest case, a base *Hypertext Markup Language* (HTML) file describes the page layout and refers to embedded objects. The page is retrieved using multiple *Hypertext Transfer Protocol* (HTTP) requests, potentially involving several servers. The term *Web transaction* refers to the sequence of requests that starts when the user sends the first request and ends when the last object is retrieved.



```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: text/html
Connection: keep-alive
...
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: ...
```

Figure 3.2: Client-server document retrieval by HTTP over TCP

The *Hypertext Transfer Protocol* (HTTP) [RFC 2616] is a stateless request-response protocol. The first steps of a Web page request are sketched in Figure 3.2. After the DNS resolution of the *Uniform Resource Identifier* (URI), the client establishes a TCP connection to the server and sends a HTTP request. The document is then delivered by the server. The subsequent interactions depend on the HTTP version. In version 1.0, the client fetches each object over a separate TCP connection. This is inefficient if a Web page refers to many objects. HTTP version 1.1 supports *persistent connections*, i. e., subsequent requests can share the same TCP connection. The use of persistent connections can significantly outperform HTTP version 1.0 [88, 165]. A further feature of HTTP version 1.1 is *pipelining*. With this extension, multiple requests can be sent simultaneously, but it is not widely used. Instead, Web browsers use multiple persistent connections to overcome idle times. According to [RFC 2616], a client should not maintain more than two connections to a server or proxy. State-of-the-art Web browsers open between 2 and 6 concurrent TCP connections to a server. Studies revealed that the optimal number is within this range [65, 45].

Advanced Web applications do not query static objects only. During the last decade, a vast amount of powerful Web technologies have emerged. Additional application logic can be executed either at the server or at the client. Server-side technologies for dynamic Websites can use various script or programming languages or complete Web application frameworks. There are also several options to execute code inside Web browsers using the *JavaScript* programming language. Furthermore, browser plugins are widely used to integrate vector graphics and multimedia clips in Websites. A further trend is that data is encoded in *Extensible Markup Language* (XML). It is used for instance in combination with *Asynchronous JavaScript and XML* (AJAX), which realizes asynchronous communication with a server without reloading a page.

These new Web technologies are in particular used by emerging *Rich Internet Applications*. They are client-server applications that run within a Web browser, but they have many features and functions like classic desktop software, including responsive user interfaces and asynchronous communication without user interactions [126]. Their key advantage is that no installation and management of local software is required. They are universally accessible through the Web. But the dependence on the network also results in limitations. Due to the frequent communication, Rich Internet Applications typically require broadband Internet connectivity.

Another trend is the use of *Web services*, i. e., mechanisms for the interoperation amongst separately developed, distributed applications. The *Application Programming Interface* (APIs) can be realized by the lightweight *Representational State Transfer* (REST) method, which adapts the semantic of HTTP, by *Remote Procedure Call* (RPC) middlewares, or by additional messaging layers such as the *Simple Object Access Protocol* (SOAP) (see e. g. [149]).

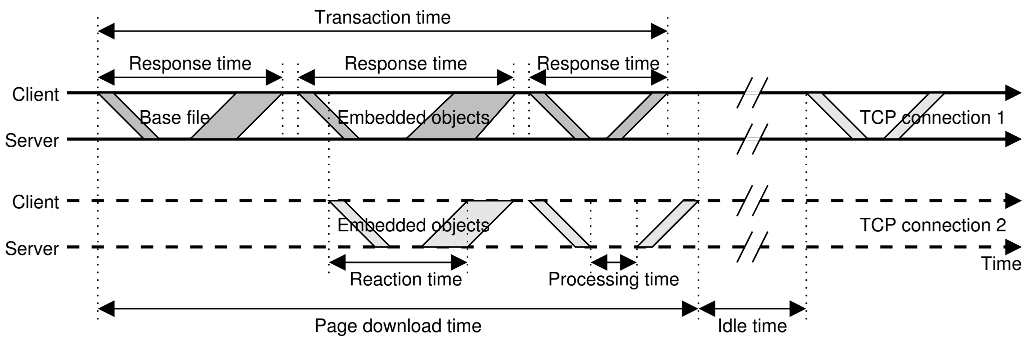


Figure 3.3: Structure of a Web transaction composed of several requests

3.1.3 Performance requirements, metrics, and service level agreements

3.1.3.1 Measurable performance metrics

Performance is one of the most important non-functional aspects of any system. Responsiveness, availability, and scalability are very important properties of Web applications. The assurance of these properties is essential in order to avoid user dissatisfaction, loss of revenue, or productivity, in particular in e-commerce and other mission-critical interactive Web applications, such as *Enterprise Resource Planning* (ERP). Several metrics can characterize the performance of interactive applications. They are introduced in the following. There are also other important non-functional characteristics, such as dependability, robustness, usability, security, manageability, or costs. They are not specifically considered in this section.

In general, *performance* can be defined as a measure how well a system accomplishes its assigned task. There are various *measurable* metrics that evaluate the user-perceived performance of interactive Web applications [151]: The *availability* measures the percentage of time customers can access an applications. There are various other related availability metrics, such as the *service access success rate*. Availability objectives typically depend on the type of application. For instance, a "five-nine" availability of 99.999 % requires a maximum downtime of five minutes per year. An important system performance metric is the *request throughput* or *transaction throughput*, which is defined as the number of request or transactions that can be served per second, respectively. For user interaction the key performance aspect is the responsiveness or timeliness, which characterizes the speed of a system as seen by the user.

The responsiveness can be quantified by the *response time* T_{resp}, which is the elapsed time between the sending and the complete reception of the response (also known as "time to last byte"). One part is the *reaction time* T_{react} that refers to the time period between initial stimulus and the first indication of a response ("time to first byte"). The *transaction time* T_{trans} measures the completion time of a transaction. For Web applications using only one TCP connection this corresponds to the *page download time* or *page loading time* T_{page}, which consists of the download time of the base file and all embedded objects. The differences between these metrics are highlighted in Figure 3.3. It is also possible to normalize the download time in a "fun factor" that considers the object sizes and link speeds [46]. Another metric is the *flow-completion time* T_{FCT}, i. e., the time between sending the first packet of a flow and receiving the last packet [56]. There have also been efforts to isolate the *network contribution to transaction time* [G.1040].

The transaction time is the sum of many delay components [151]: Communication delays include the data transfer times, the network latency, queueing and processing delays, and transport protocol delays caused, e. g., by retransmissions or by congestion control. Waiting and processing delays can also occur in the server at various stages, and again include communication delays if further servers or databases must be queried. Finally, there are local processing times in the client software, e. g., for parsing, rendering and visualizing the content. A significant share of the total transaction time can be caused by processing delays inside servers. For example, a study [16] found that Web servers can contribute to up to 80 % of the response time, in particular if the load is high. Similar results have also been reported for ERP systems [153].

3.1.3.2 Derived performance metrics

Certain system design objectives cannot be measured directly and, if at all, only be quantified by *derived metrics*. An example is *dependability*, which encompasses attributes of availability, reliability, safety, confidentiality, integrity, and maintainability. Another performance-related system requirement is *scalability*. Scalability can be defined as the ability to operate efficiently and with adequate performance over a given range of configurations [107]. Intuitively, a scalable system is able to support a large and potentially fast increasing number of users. Scalability refers both to the functional requirement that a system must be extensible in size, as well as to the non-functional aspect that increased system loads must be handled. Scalability is also closely related to cost: A system does not scale if the additional cost of coping with a given increase of the workload or if the size of the system is excessive. As a consequence, it is difficult to quantify scalability in a measurable metric. Different scalability metrics have been developed for multiprocessor problems, in particular the *fixed size speedup* (Amdahl's law) and the *fixed time speedup* (Gustafson's law). However, such simple metrics cannot be applied to complex distributed systems. A more sophisticated effort can be found in reference [107], which proposes a derived metric based on the cost-effectiveness: The *productivity* is a function of the system's throughput, its mean response time and the running cost per second. A *scalability metric* relating systems at two different scales is then defined as the ratio of their productivity figures. However, such derived scalability metrics are only partly applicable in practice.

3.1.3.3 User requirements and service level agreements

Interactive applications require timely access to information. This design target is well paraphrased by a statement of Odlyzko: "The purpose of data networks is to satisfy human impatience" [167]. The impact of response times on users has been studied for a long time. In 1968, a study described three threshold levels of human attention [155]: A response time of 0.1 s is viewed as instantaneous. If it is less than 1 s, it is fast enough for a dialog interaction. Furthermore, response times must stay below 10 s to keep the user's attention. These fundamental constraints also apply to modern Web applications. Many studies confirm that long Web transaction times lead to user dissatisfaction. For instance, a large portion of customers shopping in the Web waits no more than 4 s for a page to render [108]. The *abort probability* of search requests is typically 60 % when the response time is $4-6$ s, and up to 95 % if it exceeds 6 s [151]. Similarly, average response times beyond $1-3$ s are considered as unacceptable in enterprise systems [153]. Achieving a reasonable user-perceived performance requires to keep response times below these limits. Some Websites apparently even suffered from a loss of revenue after the Web page loading times had increased by some hundred milliseconds only [75].

Due to the emergence of more and more commercial Web applications it is necessary to define and guarantee such performance service level objectives in *Service Level Agreements* (SLAs).

```
<ServiceLevelObjective name="slo1">
  <Obliged>ACMEProvider</Obliged> [...]
  <Expression><Implies><Expression><Predicate xsi:type="Less">
    <SLAParamter>Transactions</SLAParameter><Value>10000</Value>
  </Predicate></Expression><Expression><Predicate xsi:type="Less">
    <SLAParamter>AverageResponseTime</SLAParameter><Value>0.5</Value>
  </Predicate></Expression></Implies></Expression> [...]
</ServiceLevelObjective>
```

Figure 3.4: Example for a Web Service Level Agreement encoded in XML (taken from [52]). The service provider `ACMEProvider` guarantees that the parameter `AverageResponseTime` is less than 0.5 seconds if the value of `Transactions` is less than $10,000$.

Application-level QoS targets typically cover availability and throughput targets. Response time guarantees are typically expressed in form of a maximum average value or the 90 % quantile. There have been several attempts to standardize description frameworks for SLAs in Web environments. An example for such a Web service SLA parameter specification is given in Figure 3.4. Such standardized description languages would also be useful for the dynamic negotiation of service objectives and for Web service composition [196].

3.1.4 Broadband interactive applications

3.1.4.1 Definition

A key hypothesis of this work is that the trend towards higher bandwidths enables a new class of applications with delay-sensitive user interactions. This application type can be located in the center of the application portfolio of Figure 3.1. The author is not aware of a consistent terminology and therefore suggests the following definition for this class of applications:

> **Definition 3.1 (Broadband interactive application)**: A broadband interactive application is a responsive networked application that requires fast and reliable transport of potentially large amounts of data. Typical representatives are network-challenging client-server applications that interact by bursty request-response communication and significantly benefit from timely arrival of messages, even if there is no deterministic delay deadline.

This definition is more precise than several other related nomenclatures: The term *soft real-time application* is frequently used to classify QoS requirements (e. g., in [62]). However, the expression *real-time* is misleading since it is frequently used in a context where on-time arrival of messages is more important than reliable data delivery. Another related term is *high-priority transaction service* [G.1010], which should “provide a sense of immediacy to the user that the transaction is proceeding smoothly, and a delay of no more than a few seconds is desirable.” While this description clearly has a similar motivation, it lacks the requirement of broadband connectivity. The same difference applies to the *excellent effort* class [802.1D], which is described as “the best effort type services that an information service organization would deliver to its most important customers”. The probably closest related concept is the DiffServ *low-latency data service class* [RFC 4594] that targets at “elastic and responsive typically client-server-based applications”. However, [RFC 4594] emphasizes the aspect of short-lived flows. Broadband interactive applications do not necessarily only transport small amounts of data.

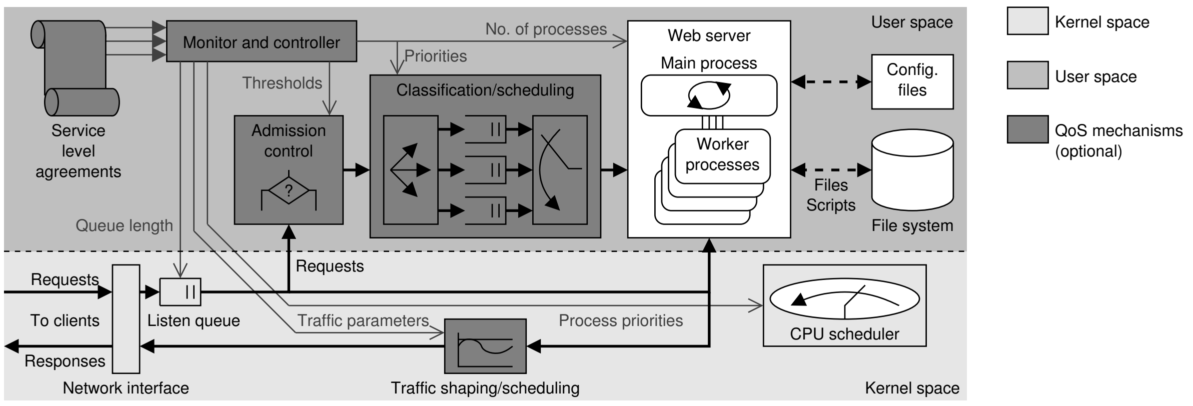


Figure 3.5: Multi-process architecture of a Web server and optional enhancements for application-level QoS provisioning

3.1.4.2 Examples

Broadband interactive applications exchange potentially large amounts of structured data, such as XML documents, still images, or other voluminous data. This type of application might have some desired delivery deadline or a target transmission time, but they typically do not require a deterministic guarantee, and they may also have own adaptation mechanisms. Such use cases can be found in several emerging application domains, including Rich Internet Applications, Enterprise Resource Planning, Grid Computing, Cloud Computing, and potentially also in certain financial and telemedicine applications. For instance, financial brokerage applications require that orders are transported with end-to-end latencies less than 50 ms. One specific focus in this work are emerging *virtual worlds*, i. e., 2D maps, virtual reality and 3D Web applications that combine voluminous high-resolution images with three-dimensional data and other context information. The paper [207] shows that online 3D visualization applications have characteristics that correspond to the definition of broadband interactive applications (see also Section 7.2). Recent Web traffic measurements [209] focusing on new AJAX applications also reveal the traffic characteristics attributed to broadband interactive applications, i. e., some Websites require frequent, large HTTP transfers. The traffic is much more bursty than the one of classic Web applications and less deterministic than multimedia streaming. Due to the ongoing trend towards Rich Internet Applications, Cloud Computing, 3D Web, and other new network-challenging traffic, one can expect the emergence of further broadband interactive applications in near-term future. The following sections review existing application-level techniques to support such applications.

3.2 Existing mechanisms for performance improvement and assurance

3.2.1 Server mechanisms

3.2.1.1 Server architecture and functions

A Web application may be served by a single server or by a complex *Web system* that is composed of several tiers of servers, e. g., Web servers, application servers, and database servers. A Web server itself is a complex software system that can be realized by different software architectures, which range from one single process to the parallel execution of multiple processes and/or threads. For example, the widely used Web server “Apache” uses by default one main process that assigns requests to a pool of worker processes, as shown in Figure 3.5. Web systems often also include *Database Management Systems* (DBMSs) that provide efficient, durable and structured storage of information. They can be queried using a high-level query language such as *Structured Query Language* (SQL).

The *performance tuning* of a Web system can be a complicated task, in particular if databases are involved [228]. Traditionally, Web servers handle incoming requests by an FCFS mechanism and use a trivial admission control: When the number of enqueued requests exceeds a predefined threshold, additional incoming requests get dropped. These mechanisms, which basically correspond to an application-level *best effort* service, provide reasonable performance if sufficient server processing capacity is available (*rightsizing*).

3.2.1.2 Application-level performance differentiation and assurance

A proper capacity planning alone is not sufficient to ensure that performance objectives or SLAs are met. The overprovisioning of server capacity is expensive, in particular due to energy consumption. Public Web applications are also subject to enormous demand variations, e. g., due to "flash crowd" effects. Network QoS mechanism only do not solve this problem. Compliance with SLAs also requires *application-level QoS mechanisms*. They can be subdivided into *differentiation* mechanisms and *assurance* mechanisms and require support in servers.

A server that provides differentiated service is able to distinguish between different priorities of requests, e. g., between a premium and a basic class, which may be subject to different SLAs [52]. Differentiation requires mechanisms for request classification and resource control. As shown by the optional building blocks in Figure 3.5, the differentiation can be realized in the server software and/or in the network stack, i. e., inside the *Operating System* (OS). Kernel-level differentiation mechanisms include admission control within the TCP listen queue [236], prioritization of certain server processes by the OS scheduler [7], and outgoing traffic shaping [83]. At the application layer, both class-based admission control and differentiation can be realized when the requests are processed [7, 21]. The number and priority of worker processes can be controlled [7]. An optimization may also be possible by a size-based scheduling algorithm, such as *Shortest Remaining Processing Time* [83]. The classification of incoming HTTP requests and the mapping to classes can either use network parameters only, i. e., IP addresses and TCP port numbers, or it can consider application information such as the URI or cookies [21, 236].

In particular for commercial Web applications it would be desirable to provide a predictable service that is independent of the demand and isolated from other services. Such application-level QoS assurance mechanisms are challenging since complex software systems share many resources without providing control over their allocation. One option are controllers that monitor the end-to-end application performance and automatically adapt the configuration and allocation of server resources [243]. Of course, stringent deadlines cannot be met if they are smaller than individual system response times [196]. Another fundamental challenge is that application-level QoS mechanisms must not only be realized in the endsystems, but also interact with the network. Guarantees for application throughput and/or response times require the availability of corresponding network resources. Therefore, application-level QoS targets must be mapped to network layer QoS mechanisms. In theory, such a mapping is possible [229], but mature technical solutions are hardly available [62]. A simple alternative to application QoS assurance mechanisms is presented in Section 7.1.

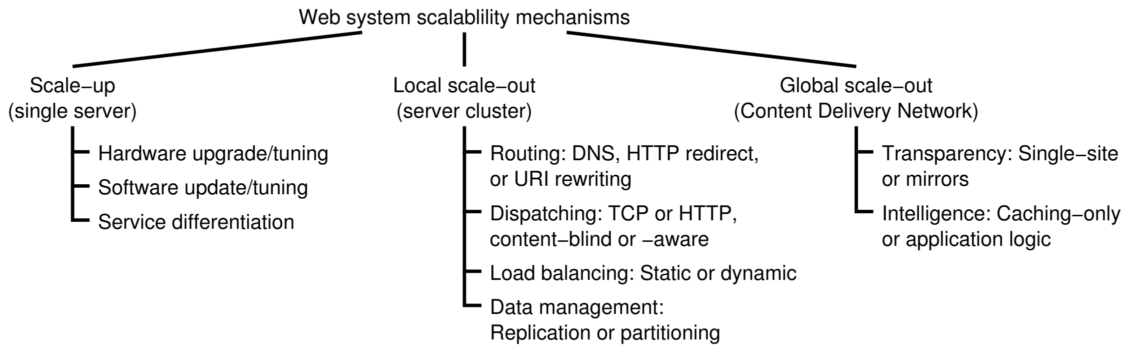

Web system scalablility mechanisms
Scale−up
(single server)
Hardware upgrade/tuning
Software update/tuning
Service differentiation
Local scale−out
(server cluster)
Routing: DNS, HTTP redirect, or URI rewriting
Dispatching: TCP or HTTP, content−blind or −aware
Load balancing: Static or dynamic
Data management: Replication or partitioning
Global scale−out
(Content Delivery Network)
Transparency: Single−site or mirrors
Intelligence: Caching−only or application logic

Figure 3.6: Classification of alternatives for building scalable Web systems

3.2.2 Distribution mechanisms

3.2.2.1 Building scalable distributed Web systems

As the utilization of a Web system grows, the processing demands can exceed the capacity of single servers and mandate system enhancements. The main architectural alternatives for building scalable Web systems are classified in Figure 3.6. The simplest choice to improve the performance of a Web system is the *scale-up* strategy, which can be subdivided into speeding up or replacing components (hardware scale-up) and tuning the software (software scale-up). If certain limits are exceeded, improving the processing power of a single server gets more and more complex and expensive. Then, distribution is an alternative. This *scale-out* strategy splits the load among several nodes. In case of *local scale-out*, the nodes reside at a single location. They form a *Web cluster* that appears to the users as a single system. *Global scale-out* refers to nodes that are located at different geographical locations (*distributed Web system*). Global scale-out services in the Web are provided by *Content Delivery Networks* (CDNs). The following sections review these technologies along the lines of a comprehensive survey [41].

3.2.2.2 Web clusters

A Web cluster is a collection of servers at a particular site that are managed by a single administrative entity. It is also referred to as a “Web farm”. Unlike in distributed Web systems, the multiple servers are not necessarily visible to users. The high-level structure of a Web cluster is depicted in Figure 3.7. A cluster consists of servers, one or more protocol mechanisms to spread client requests among the nodes, and, if necessary, one or more devices that decide

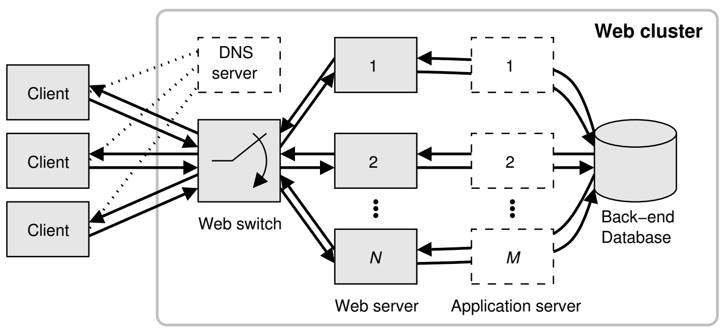


Figure 3.7: Fundamental architecture of a two-tier/three-tier Web cluster

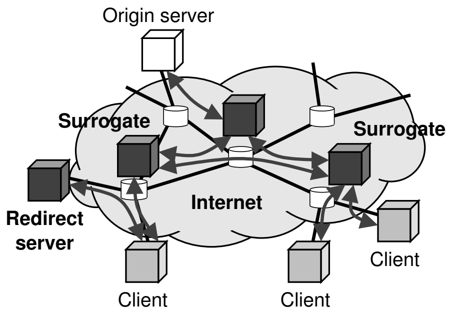


Figure 3.8: Basic structure of a content delivery network

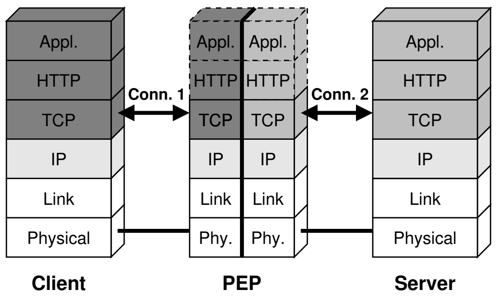


Figure 3.9: A performance enhancement proxy splitting a TCP connection

to which server a request is assigned. The component that realizes this dispatching is also named *Web switch* or *load balancer*, and it can be realized as a transport layer function ("layer-4 Web switch") or at application layer ("layer-7 Web switch"). The dispatching policy can be content-blind or content-aware, and there are various further degrees of freedom concerning its realization. Possible mechanisms include the HTTP redirection mechanism, URI rewriting, and triangulation techniques. Alternatively, requests can be routed by DNS redirection, i. e., the domain name servers dynamically return a different set of addresses according to policies.

The system architecture of Websites with dynamic content often consists of two or three tiers: In the *front tier*, Web servers accept requests and return formatted responses. State is typically stored in back-end DBMSs (*data tier*). In a three-tier architecture, there are also application servers that realize application logic (*middle tier*). Two design choices exist for the data management and content placement: If *replication* is used, the system consists of replica clones that all have the same software and data and that all can serve a request. *Partitioning* divides the data among the nodes so that a request must be directed to the appropriate partition.

Web clusters have few *scalability limits*. Large-scale clusters can be built out of commodity computer hardware in a cost-effective way [17]. A multi-tier architecture is also inherently scalable in the front and middle tier, since the servers are mostly stateless and processing can easily be parallelized. It is typically more difficult to scale the back-end tier because of the complex transaction processing mechanisms. The most important scalability limits of clusters with thousands of nodes are power supply and cooling.

3.2.2.3 Content Delivery Networks

Web content delivery at global scale is supported by *Content Delivery Networks* (CDNs), which are distributed Web systems that are widely used by popular Websites [170]. CDNs are overlay networks that consist of *surrogate* servers, which are typically located in the points of presence of ISPs. The fundamental principle of CDNs is distributed caching: CDNs replicate content in the surrogates. As illustrated in Figure 3.8, CDNs ensure that user requests are served from a topologically close site. However, many technical details of commercial CDN platforms such as replica and content placement, the request routing strategies, or consistency enforcement are not published, even though some aspects can be derived by reverse engineering [224].

CDNs push Web content towards the network edges. This has several advantages: First, the caches reduce the overall network traffic. Since the data is topologically closer to user, the risk of traversing congested WAN links or peering points is smaller. Second, CDNs reduce the network latency and thereby also Web transaction times compared to obtaining content directly from the origin server. Third, CDNs add redundancy and thereby improve the reliability of content delivery. Finally, unlike a dedicated infrastructure, the load balancing in CDNs can exploit the multiplexing gain of serving many Websites.

The traditional CDNs technology is challenged by new dynamic content in the Web. User-generated content, multimedia streams, and Rich Internet Applications result in much more complex client-server interactions. Furthermore, application service providers increasingly offer processing power and storage capacity to end users (*cloud computing*). For such new applications a pure replication or caching of static files is less effective. It is an open question whether the CDN technology will indeed be able to keep up with these trends.

3.2.2.4 Middleboxes

Another approach to improve the application performance are middleboxes that realize a *Performance Enhancement Proxy* (PEP) [RFC 3135]. There are many different types of PEPs, which are also known as *WAN optimizers*, *WAN accelerators*, *HTTP proxies*, etc. A common characteristic is the interception of the communication between the two endsystems, either at the transport layer (*TCP PEP*) or at the application layer (*Application Level Gateways*). A TCP PEP intervenes in the TCP protocol operation, e. g., by adjustments of the TCP receive window or by realizing a transparent proxy that splits the end-to-end connection (see Figure 3.9). An important mechanism at application layer is caching. Caching can be realized by an HTTP proxy, but there are also PEPs that try to optimize other application protocols. Several other application-specific mechanisms can be used to boost the performance, including compression, protocol modifications, traffic conditioning, and the usage of proprietary protocols between gateways. In particular in mobile networks, proxy-based *transcoding* is also used in order to adapt Websites to mobile devices, e. g., by reducing the resolution of images.

The performance of Web applications can be improved by PEPs in various ways. Transparent TCP proxies split the end-to-end path into two connections that each have a smaller delay. Due to the TCP protocol mechanisms (see Section 4.2.1.2), smaller delays speed up the error recovery and the congestion control. When a client communicates with an HTTP proxy, it can maintain persistent TCP connections to that proxy instead of opening many short connections to different Web servers. This avoids the overhead of connection setups and the connections are less likely to be limited by the TCP Slow-Start [88]. Studies have shown that properly configured PEPs can reduce Web page loading times, in particular in cellular networks [162].

However, PEPs violate the end-to-end design principle of the Internet and may hinder application protocols that require end-to-end connectivity. They require additional processing and storage capacity and have limited scalability. A PEP is also an additional source of errors and may become a single point of failure. Since the end-to-end communication is broken, end users have to trust that the PEP functions are correctly implemented. Moreover, PEPs are not compatible with encryption, digital signatures, and mobility mechanisms in the IP layer. This is why [RFC 3135] disadvises proxies that automatically intervene in all communication.

3.2.3 Client mechanisms

The main client for Web applications is the Web browser, which typically consists of four main components: The GUI, the browser control, the parsing engine, and the rendering engine. Web

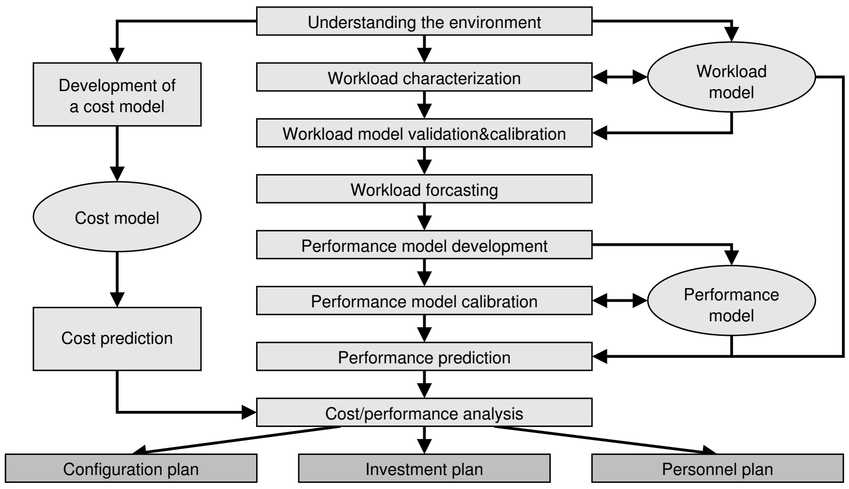

Understanding the environment
Development of a cost model
Workload characterization
Workload model
Workload model validation&calibration
Workload forcasting
Cost model
Performance model development
Performance model calibration
Performance model
Cost prediction
Performance prediction
Cost/performance analysis
Configuration plan
Investment plan
Personnel plan

Figure 3.10: Capacity planning workflow (according to [151])

browsers can optimize the performance by several techniques: Local caching is widely used to avoid repeated retrieval of popular objects. The initial-screen response time can be reduced by optimizing the request order and intelligent TCP connection management [45]. In addition, on-the-fly compression can result in significant savings [165]. The performance can also be increased by proactive *pre-fetching* of documents at the cost of an increased network load [209]. Finally, the client can negotiate suitable content formats with the server in order to perform *content adaptation*, as realized for instance in [141].

3.3 Performance evaluation methodology

3.3.1 Role of performance evaluation

Performance evaluation is a scientific method in the field of systems engineering that analyzes, predicts, or optimizes quantitative parameters of a technical system. It can either be applied to real systems or to a *model*, which is an abstract description of a system encompassing its components, their interaction, and the interactions with the environment. The behavior of a complex system also depends on its internal configuration, which might not be entirely known. Building models that accurately represent a complex system is a challenging task.

Several scientific disciplines focus on specific aspects of performance evaluation: *Performance analysis* investigates the quantitative characteristics of a system, e. g., by measurements, *performance tuning* improves the system, and *performance prediction* forecasts the behavior outside the known range of operation. Their combination is a part of the *performance engineering process*. In practice, the development processes often focus on functional aspects and consider performance issues only at a rather late stage (“fix-it-later” approach) [216]. This workflow typically also applies to the development of networked applications. Therefore, achieving reasonable performance levels is often seen as a *capacity planning* and *resource dimensioning* problem only. Capacity planning is an iterative process that is realized at several different time scales. As shown in Figure 3.10, it encompasses different steps and both performance and workload models, which are further detailed in the next sections.

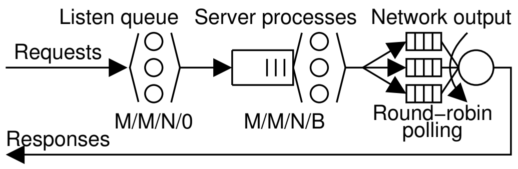


Figure 3.11: Simple queueing model of a Web server according to [183]

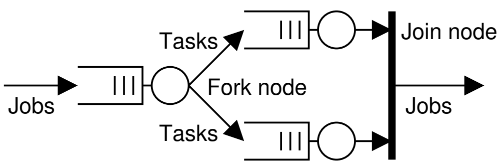


Figure 3.12: Example of an open fork/join queueing network

3.3.2 Methods for performance modeling, prediction, and measurement

3.3.2.1 Queueing theory models

Queueing theory provides a set of well-established mechanisms to analyze the performance of computer and communication systems [86]. The simplest model component are single-server queues, such as the M/M/N model with *Poisson arrivals*, exponential distributed service times, FCFS discipline, and either infinite (M/M/N/∞) or finite buffers (M/M/N/B). For such models, queueing theory provides elegant closed-form equations for several performance metrics.

Queueing theory can be used to model interactive applications. For instance, the internal structure of a server can be modeled by a queueing network as shown in Figure 3.11. Under certain constraints (e. g., “BCMP networks” with product-form solution) either closed-form solutions or very efficient numerical solution methods exist [27, 86]. A specific challenge are concurrent processes. They can be modeled by *fork/join queueing networks*: As depicted in Figure 3.12, a job is split into several tasks that are distributed over several parallel service units, and it can only be finished when all tasks have completed service. This model is not part of the classic queueing theory and can only be analyzed under certain conditions. Exact results are known for some special cases [163], while more complex models can only be analyzed by approximations (e. g., [119, 196]) or by decomposition approaches [13].

The assumptions required by queueing theory are often not realistic. The performance of applications results from a complicated interplay between a variety of components, such as hardware platform, server software, operating system, protocols, network behavior, workload characteristics, etc. Because of this complexity, it is difficult to derive queueing models for Web applications, and even if it is possible, the solutions are typically numerically non-trivial.

3.3.2.2 System performance engineering methods

Another method for performance prediction is *software performance engineering*, which advocates the integration of performance analysis into the software development process in an early stage [216]. The objective is to predict performance based on high-level structure and behavior models, such as *Unified Modeling Language* (UML) diagrams or *Specification and Description Language* (SDL) message sequence charts. In addition to queueing or layered queueing networks, there are several other methods for model-based performance prediction that use stochastic petri-nets, process-algebra, or simulation [216]. An often underestimated problem is the parametrization of such models. All known modeling approaches require the knowledge of parameters that can only be obtained from a prototype or a complete implementation [216], i. e., when the software and hardware platform is already available. This model calibration problem is hardly addressed by literature on software performance prediction – apart from [135].

An estimation of the internal performance characteristics can be realized either by *instrumentation*, i. e., by additional performance monitoring instructions in the code, or by *profiling* within a debugging environment. An example for the application of both methods can be found in [238]. Such *white box tests* require a detailed knowledge of the internal system realization. Therefore, white box tests are often either technically difficult or very intrusive.

In practice, the performance of software and hardware systems is often measured by *benchmarks*. Benchmarks consist of systematic measurements of the performance of a *Device Under Test* (DUT), using a well-defined environment and workload. Benchmarks are *black box tests*, i. e., there are typically no or only few assumptions about the inner structure of the system. One or several load generators offer stimulus to the DUT and measure its response. A typical benchmark is a *stress test* that determines the capacity of a system. There are domain-specific standardized benchmark suites that compare the performance of Web servers, databases, etc. Due to the precise description of the setup and scenarios, such benchmarks facilitate the comparison of performance of different software and hardware systems.

Due to the limitations of other theoretical methods, this thesis studies the application performance by a combination of black box and white box tests.

3.3.3 Traffic and workload modeling

3.3.3.1 Model types

There are many models for the statistical properties of *network traffic* and *application workloads*. Computer network traffic is known to be self-similar, i. e., there are long-term correlations in the packet arrivals [132]. Therefore, Poisson processes are not well suited for modeling traffic at packet level [176], which is known since the mid-1970s [54]. To some extent, the self-similarity can be attributed to the behavior of the TCP congestion control [132, 176, 66]. Traffic models for aggregated traffic are hardly related to performance of individual applications. Therefore, the reader is referred to reference [65] for further details on traffic models.

Application workload models can be classified into *synthetic source-level models* and *trace-based workload models*. Synthetic source-level models characterize the workload by a mathematical model. The model parameters, such as *Inter-Arrival Times* (IATs) or *object sizes*, are described by analytical distribution functions (*parametric models*) or by empirical distribution functions (*non-parametric models*). Synthetic models are specific for an application type or a mix of applications. They are typically composed of different levels: For instance, [151] suggests a user level (user sessions), an application level (system functional aspects), a protocol level (e. g., HTTP), and a network level. But there are also other possibilities [65]. Synthetic source-level models have been developed for many popular Internet applications, in particular in the WWW [65]. They are parametrized by the result of large empirical measurements. The development and calibration of application-specific workload models is a complex process and must be repeated each time the communication behavior changes. A further drawback is that the analysis of measured traffic requires knowledge of the application protocols. As a consequence, most existing synthetic source-level models only address few applications. Many published models also lack behind the real workload characteristics in the Internet.

In the last decade, the *Scalable Uniform Resource Locator Reference Generator* (SURGE) model and tools [15] have been a kind of *de facto* standard for Web workload modeling. SURGE is a set of benchmark tools that imitate a fixed population of Web users modeled by on/off processes. The workload is generated from empirically derived distribution functions for the file

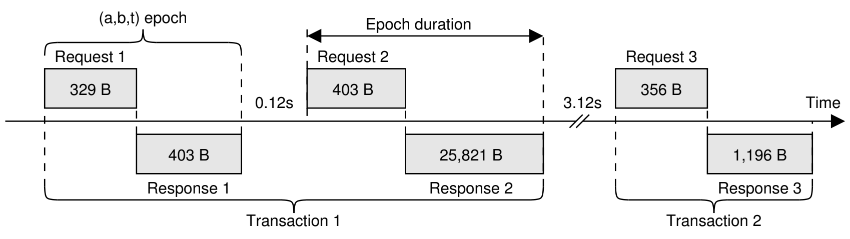


Figure 3.13: Model for the message exchange in a TCP connection (adapted from [244]). In *a-b-t* syntax this corresponds to the vector $[(329,403,0.12),(403,25821,3.12),(356,1198,0)]$.

size distribution at the server, the request size distribution, the relative file popularity, the embedded file references, the temporal locality of references, and idle periods (think times) of individual users. The parameters used by the latest version of the SURGE software slightly differ to the original model [15].

An alternative to synthetic source-level models is *trace-based workload modeling*. The fundamental principle is to reconstruct application activities from captured network traffic. Replaying the traces then reproduces the essential patterns of application read and write operations. In reference [244], such a source-level characterization is developed for TCP connections: Each TCP connection is represented by a *connection vector*, which is a series of individual data exchanges (*epochs*) between the TCP connection initiator and acceptor. The data units model application messages. The application messages are separated by time intervals that represent application processing times or think times. In each connection, the pattern of application message exchanges is represented by a connection vector $\mathbf{V} = [\mathbf{E}_1, \mathbf{E}_2, \ldots, \mathbf{E}_j]$, which consists of epochs $\mathbf{E}_i = (a_i, b_i, t_i)$. a_i is the size of the i-th message sent from the connection initiator to the acceptor, b_i the size of the i-th message in the reverse direction, and t_i is the think time between the receipt of the ith response and the transmission of the $(i+1)$-th request. An example is shown in Figure 3.13. The *a-b-t* connection vector notation can also model application protocols that omit one of the messages during the exchange by $(a_i, 0, t_i)$ or by $(0, b_i, t_i)$. The model can also be extended to 4-tuples of the form $\mathbf{E}_i = (a_i, t_{a,i}, b_i, t_{b,i})$ in order to include server response times. The workload description by connection vectors can be performed without any *a priori* knowledge about the application protocols. It can also model the aggregated behavior of many different applications. Yet, a drawback of trace-based workload modeling is that it is hard to obtain insight into the system behavior and specific application performance metrics.

Workload models are implemented by *load generators*. There are two different load generation principles: *Open-loop* generators inject data according a model of the arrival process, typically assuming an infinite number of sources. Open-loop generators can specify an offered network load, but they are insensitive to the network characteristics. In contrast, *closed-loop* models take feedback loops into account, e. g., the impact of congestion control. As the network load depends on the control loops, it is impossible to define an offered load. The trace-based TCP workload modeling [244] is a hybrid approach: Each connection realizes a closed-loop model. Still, the arrival process of connections is described by inter-arrival times, i. e., an open model.

3.3.3.2 Known Internet and WWW workload characteristics

The total workload in the WWW is subject to large fluctuations during one day and also during the week with a sleep-wake pattern per day and less volume on weekends [70, 152, 231]. A general trend is that the mean value of the transfer sizes increases, in particular for new Web ap-

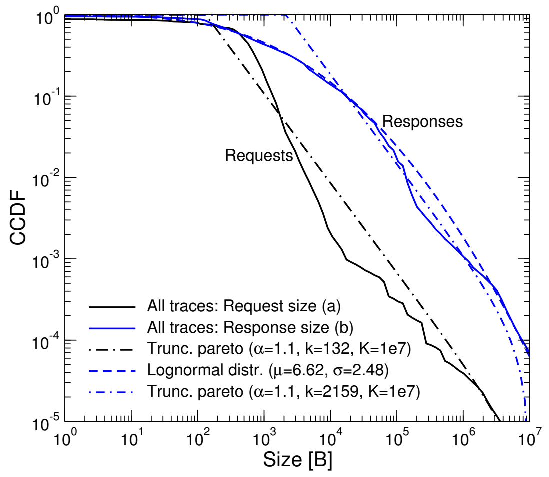


Figure 3.14: Observed request and response size distribution in the used workload traces

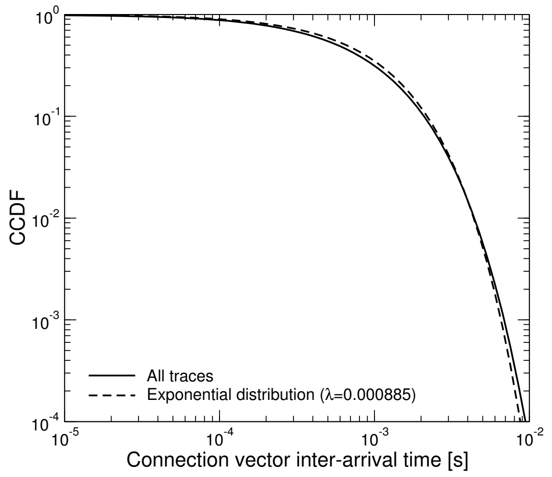


Figure 3.15: Observed connection IAT distribution in the used workload traces

plications [209]. Concerning the distribution, some studies observe heavy-tailed distributions, but there are also reports for other long-tailed shapes, in particular lognormal distributions [55]. A common approach is to model file sizes by combined distributions [15]. It is agreed that the distribution of requested files follows Zipf's law [15]. Furthermore, there is evidence that user session arrivals can be modeled by Poisson processes [176, 66, 20]. Another known characteristic of the Internet is that a large part of the overall traffic is created by a small number of connections ("elephants"), while the vast majority of connections has only a short lifetime ("mices") [79]. Data about the internal performance of Web systems is hardly published, and the available studies report different characteristics. For example, [153] observes lognormal response time distributions in ERP systems. Some measurements of e-commerce workload [231] argue that backend response times can be best described by a pareto distribution, and there are also further indications of heavy-tailed server response times [65, 196]. Existing studies on the WWW workload are comprehensively surveyed in [65].

In the following chapters, both synthetic and trace-based workload models are used. As recommended by Andrew *et al.* [10], the trace-based workload models use data from traffic traces that are publicly available [239]. These traffic traces have been measured on a university campus in the year 2008, and the published data is already post-processed to describe TCP connection vectors. Figure 3.14 and Figure 3.15 show two exemplary Complementary Cumulative Distribution Functions (CCDFs) for two statistical properties of these traces: The application message size and the IAT of connection vectors. The response size distribution has a mean of about 14 kB and can be well approximated by a lognormal distribution. The average request size is much smaller (1 kB), and the distribution is rather unsteady. The distribution tails could also be modeled by a truncated Pareto distribution with shape factor $\alpha_{\text{pareto}} = 1.1$. Figure 3.15 reveals that the inter-arrival time of vectors is almost exponentially distributed.

3.3.4 Accurate simulation of real network stacks

3.3.4.1 Advantages and drawbacks of performance evaluation by simulation

In networking research, protocols and systems are often developed and evaluated with help of *stochastic discrete-event simulation*. This means that a simulation tool studies the implementation of an abstract model in a virtual simulation time. Simulation fills a gap between theoretical studies and experimentation: On the one hand, simulation can be used to predict the performance of systems and protocols that are too complicated for a mathematical analysis. On the other hand, a simulation study does not require a complete experimental setup. This is why simulation studies can explore new environments or architectures even if required technologies are not available yet, or if access to the corresponding systems is not available. Simulation studies can also easily scan large parameter ranges in a totally controlled environment.

However, unlike real measurements and experiments, simulation studies only explore a constructed, abstract model. Simulations are only useful if the real systems are accurately represented by this model. In the last years, it can been argued that there is a "deep crisis of credibility" of simulation and that one cannot rely on the majority of published simulation results [175]. Many simulation studies and tools lack comprehensive verification and validation efforts. *Verification* is a process to evaluate how faithfully the implementation of a model matches the developer's intent, as expressed by conceptual descriptions and specifications. *Validation* is a process that evaluates how accurately a model reflects the real-world phenomenon that it purports to represent [89]. A simulation tool must be verified and validated. It only supports the claims that are made if the simulation is (1) repeatable, (2) unbiased and not specific to a scenario, (3) rigorous, i. e., if the aspect being studied is exercised, and (4) statistically sound. If these requirements are fulfilled, simulations are a useful tool to systematically explore the parameter space of a system. However, they cannot completely substitute real measurements, which are anyway needed as a reality check and a validation of implicit assumptions.

Another serious problem is that the Internet has several properties that are difficult to characterize and thus to simulate. Floyd states that "due to the heterogeneity and rapid change in the Internet, there does not exist a single suite of simulation scenarios sufficient to demonstrate that the proposed protocol or system will perform well in the future evolving Internet" [70]. Furthermore, there is a tremendous variety of protocol implementations. Internet protocols and in particular transport protocols are only specified to the level necessary to ensure successful communication between nodes. This implies that many engineering decisions and optimizations are left to implementers [89]. Possible remedies include the standardization of benchmarks [10] and the use of simulators that accurately model real protocol implementations.

3.3.4.2 Possibilities to accurately simulate network stacks

One fundamental challenge of networking research is TCP modeling. Research on Internet application performance requires accurate and validated TCP simulators. However, as described in the next chapter, TCP is a complex protocol that is still evolving, i. e., new features are being added over time. The behavior of TCP is also influenced by other components of the operating systems, such as the memory management, the process scheduler, and interrupt handling (cf. Section 4.2.4). Many operating systems also implement specific, non-standardized protocol optimizations. Traditional network simulators neglect these aspects and use rather simple models of TCP. However, many TCP enhancements, and even subtle implementation details, significantly affect the achievable performance. As a consequence, the behavior of state-of-the-art TCP/IP stacks differs substantially from the simplified models used by most simulation frameworks.

There are different approaches how to simulate TCP/IP network stacks, which are classified in Table 3.1. The table only includes *simulation* methods. There is also a multitude of *net-*

Table 3.1: Classification of network stack simulation approaches (presented in [200])

Type	**Modeling scope**	**Real OS code**	**Appl. models**	**Examples of tools**
Abstract model	Mathematical model only	No	Greedy sources	Any simulation framework can be used
Reimplementation	Simplified protocol functions	No	Synthetic models or traces	NS-2 [148], IKR Tcplib [24]
Code extraction	Only the TCP congestion control	Small parts only	Synthetic models or traces	NS-2 TCP-Linux [241]
Stack integration	Complete TCP/IP network stack	Network stack code	Synthetic models or traces	NSC [103], OppBSD [23], Lunar [118]
Runtime emulation	Kernel or user space emulation	Complete OS with patches	Real applications	NCTUns [237], UML Simulator [8]

work emulation mechanisms, such as the Linux network emulation (“NetEm”) [90]. The main difference between simulation and emulation is that simulation tools use a virtual time.

Abstract models characterize the fundamental macroscopic behavior of the TCP protocol and its congestion control, e. g., by using fluid-flow models (see Section 4.2.1.3). All existing abstract models are limited to simple network scenarios and straightforward workload models, in particular greedy sources. In order to enable studies of the microscopic TCP behavior, many network simulators include simplified built-in TCP simulation modules. A very common simplification is to implement only selected protocol functions and to support only one-way transport. The *de facto* standard for Internet research is the *Network Simulator version 2* (NS-2) [148]. Compared to other network simulators, the TCP models of NS-2 offer a lot of features and they have been validated extensively [89]. However, there are also a number of significant limitations: NS-2 does not implement the TCP flow control, i. e., the receive window, and connection establishment and teardown is also not completely realized [148]. Other TCP simulation frameworks have similar deficits. For instance, the IKR simulation library (“IKR Simlib”) [25] includes a TCP simulation model (“IKR Tcplib”) [24, 138]. Like the standard edition of NS-2, it lacks behind real TCP implementations that already support many new TCP enhancements, such as new congestion control algorithms.

These shortcomings can be addressed by using the code of an existing, full-featured TCP implementation that originates from an open source operating systems, such as Linux or variants of the *Berkeley Software Distribution* (BSD). An obvious advantage is that such code is inherently validated and widely tested. Still, using kernel code in a user-space simulation tool requires solutions for a couple of problems [200]:

- Differences between kernel-space and user-space environments: User space tools cannot use privileged instructions and do not need many kernel subsystems, e. g., the kernel memory management. Furthermore, there are name collisions and possibly a mismatch between programming languages.

- Coexistence of multiple stack instances: Operating system code is not re-entrant, i. e., it uses global variables that prevent multiple different instances of a stack to run concurrently within the same process space.

- Simulator interface: The network and application interfaces must be adapted to the simulator. The kernel code must be forced to use the virtual time. Also, the simulation tool must be able to deal with full packets that contain real headers and payload.

Table 3.1 lists several solutions that directly execute real network stack code in simulation tools. The simplest approach is to use only small parts of the network stack. Reference [241] presents a solution that integrates the pluggable TCP congestion control modules from Linux kernels in NS-2 ("NS-2 TCP-Linux"). With this extension, NS-2 can use all TCP congestion control variants that are implemented in Linux. But there are still notable performance differences to the real Linux stack because other parts of the TCP engine still use a highly simplified realization, e. g., the loss detection module ("scoreboard") and the processing of *Selective Acknowledgments* (SACKs) [241].

The other direct code execution alternative is to integrate a complete TCP/IP stack into a simulation environment: At the time of writing, the most elaborated solution is the *Network Simulation Cradle* (NSC), which has been developed by Jansen [103]. The NSC enables the execution of the network stacks of several Linux kernel versions and different BSD variants within simulation frameworks. The NSC provides an own parser (*globalizer*) that preprocesses the operating system source code and replaces global variables and their references by elements of arrays. As a result, it requires only very limited manual modifications of the kernel source code [105]. The NSC can be integrated in the NS-2 simulator, but its core is independent of a specific simulation tool. The NSC has undergone significant validation and performance tests [103, 104].

There are also two other similar approaches: "OppBSD" [23] integrates the FreeBSD network stack in the OMNeT++ simulator. "Lunar" [118] is a user-space library built from an old Linux kernel. Unlike NSC, both approaches require significant manual editing of the kernel code, and the underlying network stacks do not implement recent TCP enhancements.

Finally, runtime emulation or virtualization techniques use a complete operating system for simulation-like purposes. "NCTUns" [237] realizes a combination of simulation and emulation. The host operating system (BSD or Linux) is modified to run in a virtual time. By means of tunnel interfaces, simulation tools can directly use the native TCP/IP protocol stack of the host system. The "UML Simulator" [8] extends user-mode Linux virtualization with an event-driven simulation engine. Both approaches can run almost unmodified applications in virtual time. However, the required special support from the host operating system limits the applicability. In addition, the simulation speed is low and the scalability in terms of stacks is very limited.

3.3.4.3 A new TCP/IP simulation framework

The investigation of the performance of Internet applications and of new transport protocol mechanisms requires accurate models of the microscopic TCP behavior. This results in a problem: On the one hand, TCP models that behave like a given real implementation have to model the implementation-specific design choices of that stack. As a consequence, the obtained results may not be universally valid. Also, the more accurately TCP is modeled, the more complex are the simulation tools, resulting in longer simulation execution times. On the other hand, simplified models can hardly approximate the sophisticated algorithms of modern network stacks. The direct code execution approach of the NSC is a reasonable trade-off between manageability and accuracy if the network scenarios are not too large.

In order to perform the simulation studies presented in this thesis, the original NSC framework has been enhanced and adapted to the IKR simulation library. The original port of NSC to the

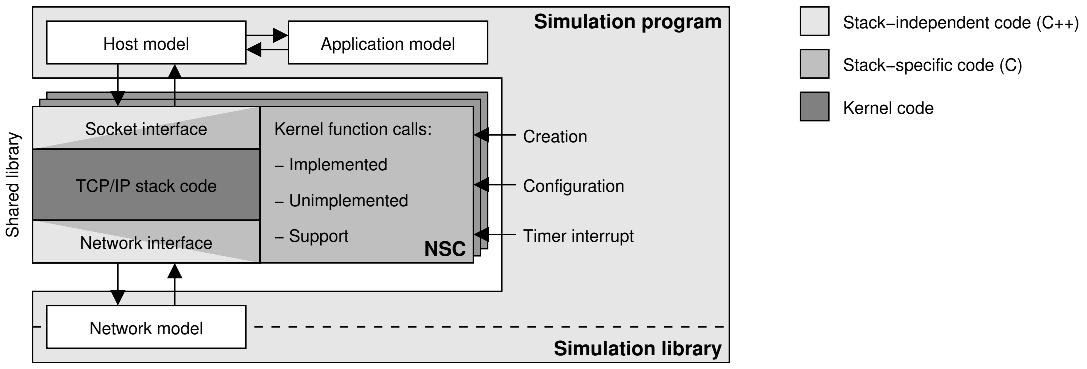


Figure 3.16: Architecture of a simulation tool with the Network Simulation Cradle (NSC)

IKR simulation library has been realized by Zeeh [253]. This implementation has also several features that are not available in NS-2, e. g., a technique to reduce the memory requirements by transporting only header parts through the simulation model. Tools within this framework have been developed by Zeeh [253], Proebster [180], and the author of this thesis.

The structure of the simulation framework with the NSC is illustrated in Figure 3.16. The core is the NSC version 3.0, which is able to simulate the network stack of Linux kernel 2.6.18, as well as other stacks. The NSC has a part that is common to all stacks, which defines several interfaces to the simulation tool. Each supported stack is complemented by implementations of these interfaces as well as re-implemented kernel functions, i. e., unused kernel functions that are replaced by user-space code. Since each network stack is contained in a shared dynamic link library, a simulation tool can easily load multiple different stacks in parallel. Beside the default Linux kernel, the NSC has also been patched by the author to run modified kernels (cf. Section 6.1.1.1). Due to the sophisticated parser only a small amount of manual work is needed to integrate a new kernel; adaptations are mainly limited to configuration interfaces.

The comparison of simulation and measurement results, which are presented later in Section 6.2, shows that the usage of real stack code results in accurate results very close to measurements in a real testbed. This is one of the main advantages compared to NS-2 and other non-validated simulators such as [24, 138]. However, this comes at the cost of longer simulation times and larger memory requirements. Furthermore, there are several issues that are difficult to solve in a simulation tool that uses real network stack code. Since kernel code is not written for a proper cleanup, it is very difficult to completely delete program objects that include a network stack. Because of similar reasons, the complete realization of the connection teardowns is non-trivial in the simulation, too. However, these issues hardly limit the practical usability of the simulation framework.

4 Fast startup congestion control mechanisms

The purpose of fast startup congestion control mechanisms is to almost immediately utilize a given path. This chapter first reviews the fundamentals of congestion control and systematically classifies the possible solutions in order to show how new congestion control mechanisms can be designed. Then, a comprehensive survey of the state-of-the-art mechanisms and the remaining open issues of the Internet congestion control is presented, which reveals that the flow startup is still a problem that is not entirely solved. The second and main part of this chapter addresses the question how fast startup congestion control could be realized. It completely reviews the design principles and classifies all known solutions. A specific focus is the analysis whether fast startup can be realized by an end-to-end congestion control, or whether network support should be used. For both cases, the author motivates and develops new mechanisms: It is shown that recently proposed end-to-end flow startup schemes must be enhanced in order to be useful in practice. Within the class of network-supported approaches, the Quick-Start protocol is extensively analyzed, and several enhancements and new algorithms are proposed. Finally, this chapter also studies radically new network-supported congestion control frameworks that originate from clean slate research projects, and it discusses their applicability compared to more evolutionary fast startup TCP extensions.

Congestion control is a very broad topic with many facets. This chapter introduces only aspects that are relevant for this work. In particular, only unicast traffic is considered. A survey of multicast congestion control issues can be found in [248]. Many issues that are only briefly mentioned in this chapter are also presented in more detail in Welzl's book [246].

4.1 Systematic classification of congestion control methods

4.1.1 Terminology and fundamentals

4.1.1.1 Congestion

There have been various efforts to precisely define the term "congestion", and the discussions did not reach agreement. A common statement is that "congestion occurs in a computer network when resource demands exceed the capacity" [100]. It is also common to consider a network congested if, due to overload, excessive queueing delays and/or packet losses occur. This definition is also used in this thesis. Keshav [115] claims that such definitions are not satisfactory because these symptoms may be due to phenomena other than congestion, and suggests a formal definition from the user's point of view: "A network is said to be congested from the perspective of a user if that user's utility has decreased due to an increase in network load". The impact on performance is also emphasized by other references, such as [Y.1221], where con-

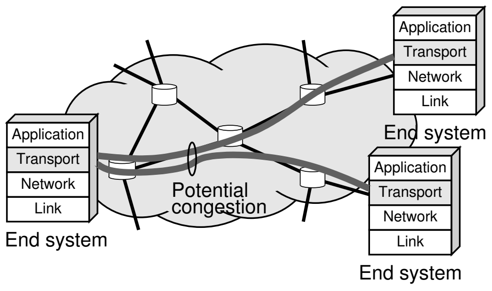


Figure 4.1: Congestion in a network caused by competing connections

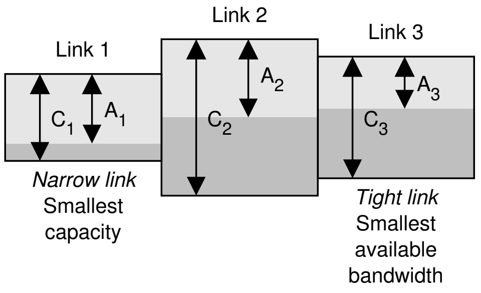


Figure 4.2: Pipe model of a three-hop path with different narrow and tight links

gestion is defined “as a state of network elements (e. g. router, switches) in which the network is not able to meet the network performance objectives and the negotiated QoS commitments for the already established flow”. Recently, Briscoe *et al.* [32] suggest a probabilistic notation and define the congestion at a resource as the probability that another packet will not be served to its requirements. Strictly speaking, resources can be *bit-congestible* or *packet-congestible* [172].

Due to the temporal multiplexing, short term load imbalances are unavoidable in packet networks and have to be corrected by buffering. If a resource gets congested, queueing delays increase, and packets must be dropped if the buffer size is exceeded. Without appropriate countermeasures, a *congestion collapse* can occur, i. e., resources are either wasted by unnecessary retransmissions or by packets that are dropped before reaching their destination [RFC 2914].

4.1.1.2 Congestion control

Congestion has to be avoided because it increases delays and wastes resources. The objective of *congestion control* is to minimize the intensity, spread and duration of congestion [Y.1221]. This requires two different functions: First, congestion control should prevent a source from sending data that will get dropped on the path; this aspect is also called *congestion avoidance* [100]. Second, it must ensure that a network remains operational when congestion occurs and react accordingly. As the name implies, congestion control is a *control mechanism*. Since the entity that governs the resource usage is not necessarily identical to the resource that gets congested, congestion control is an inherently distributed problem that requires some form of feedback and a closed control loop. Therefore, congestion control can precisely be defined as “the feedback-based adjustment of the rate at which data is sent into the network” [247]. In order to avoid and handle overload situations, congestion control mechanisms must be able to decide on the usage of resources at least to some extent. Thus, congestion control can also be understood as an “algorithm to share network resources among competing traffic sources” [172].

As shown in Figure 4.1, congestion and its resolution may affect different entities. This inherently results in fairness issues, which are discussed in the next sections. It must also be emphasized that resource management in packet networks is performed by several control loops on different time scales. The reaction time of congestion control depends on RTT of the path and is of the order of milliseconds. This is the main difference compared to other *traffic management* and *traffic engineering* mechanisms, such as routing policies or capacity dimensioning, which operate on longer time scales and often also depend on human interaction. Further control loops may also exist inside applications, e. g., by application adaptation functions or by manual user reactions. Such mechanisms typically work on time-scales longer than the path RTT, too.

In some references, the terms *congestion control* and *flow control* are used synonymously, or one is regarded as a special case of the other. In the context of transport protocols, the two terms refer to two different functions, even if they may use similar mechanisms. According to [RFC 793], flow control is defined as "a means for the receiver to govern the amount of data sent by the sender". The objective is to ensure that a sender cannot continuously transmit data faster than the receiver can absorb it. The difference to congestion control is stated precisely by Jain [100]: "Flow control is an agreement between a source and a destination to limit the flow of packets without taking into account the load on the network. It ensures that a packet arriving at a destination will find a buffer there. Congestion control is primarily concerned with controlling the traffic to reduce overload on the network. Flow control solves the problem of the destination sources being the bottleneck while congestion control solves the problem of the routers and the links being the bottleneck. Flow control is a biparty agreement. Congestion control is a social (network-wide) law." In a sliding window protocol, flow control can easily be realized by ensuring that the source's window is not larger than the free space in the sink's buffer. Congestion control is more complicated, since it involves many potentially uncooperative parties.

4.1.1.3 Network performance metrics

The purpose of congestion control is to use the capacity of a network efficiently. Defining the *capacity* of a link or network path is not always straightforward. The encoding scheme, framing, and media access results in a certain overhead and – depending on the technology – a varying capacity. Therefore, the term capacity is only meaningful if it is defined relative to a protocol layer, and it is not necessarily a constant value. In this document, resources are always described at the network layer with the following commonly used terms and metrics [179]:

- *Link capacity* r_i of link i: The maximum IP layer transfer rate is given by the maximum number of IP layer bits that can be transmitted over the link during a specific time interval. At IP layer, a link is also labeled *hop*.
- *Path capacity* r: The maximum possible transfer rate is the smallest link capacity along that path $r = \min_{1,\dots,N_{\text{hops}}} r_i$, where N_{hops} is the number of hops in the path.
- *Link utilization* ρ_i: The utilization is the fraction of the *link usage*, i. e., the number of correctly received IP layer bits during a certain interval, and the link capacity r_i.
- *Available path capacity* or *available bandwidth* v: The available bandwidth is the unused (spare) capacity on a path, i. e., $v = \min_{1,\dots,N_{\text{hops}}} v_i$, where $v_i = (1-\rho_i)\,r_i$ is the *available link capacity* of link i.

At the transport layer, further metrics are used:

- *Throughput* or *goodput* G: The data rate that an application achieves in a specific setting.
- *Bulk transfer capacity* BTC: BTC is the maximum long term average throughput obtainable by a single flow of a standard-compliant TCP connection over a path. Its value is only defined for a given congestion control algorithm, and it may be different from G.

All given metrics may be time-varying. Therefore, the time and the length of the measurement intervals are important parameters, too. As illustrated in Figure 4.2, a link i with the smallest link capacity $r_i = r$ is called the *narrow link* of a path. A link j with the smallest available link capacity $v_j = v$ is the *tight link* of the path. Furthermore, a link k can be defined as the *bottleneck link* if it has a bulk transfer capacity $BTC_i = BTC$. The *narrow link*, the *tight link*, and the *bottleneck* do not have to be identical, and their position may change over time.

4.1.1.4 Review of congestion control theory

Since the early empirical development of the congestion control mechanisms by Jain, Jacobson, and others, a significant progress has been achieved in the mathematical modeling of congestion control. Suitable methods include optimization, control theory, probability, and concepts from microeconomics [220]. An important theoretical concept are *utility functions*. In economics, utility is a measure of the relative satisfaction from consumption of various goods and services. Elastic applications can be described by a utility function $U_i(x_i)$ that is usually strictly concave, i. e., the utility increases with the available bandwidth, even if there is little evidence about the shape of the function [212].

Kelly formalized congestion control as a utility maximization problem under link capacity constraints [111, 112, 113]. In Kelly's framework, which has become a kind of standard flow level modeling of congestion control, $\mathscr{L}$ is the set of all links and $\mathscr{S}$ the set of all sources in the network. Each user $i \in \mathscr{S}$ sends packets with sending rate x_i. The aggregate link rate is $\mathbf{y} = \mathbf{A}\ \mathbf{x}$, where $\mathbf{A}$ is the $|\mathscr{S}| \times |\mathscr{L}|$ routing matrix. In $\mathbf{A}$, the entry at (i, j) is 1 if source i's route passes through link j, and it is 0 otherwise. Each link $j \in \mathscr{L}$ is assumed to have a fixed capacity r_j. Based on the level of congestion, a link price $p_j = h_j(y_j)$ can be computed. This link price information is sent back to each source with the aggregate price $\mathbf{q} = \mathbf{A}^T\ \mathbf{p}$.

Kelly formulated the congestion control problem as a static optimization and dynamic stabilization problem: The static optimization problem computes the desired equilibrium by maximizing the sum of the utility functions $U_i(x_i)$, while complying with capacity constraints of the links:

$$\max_{x \geq 0} \sum_{i=1}^{|\mathscr{S}|} U_i(x_i) \quad \text{subject to} \quad \mathbf{A}\ \mathbf{x} \leq \mathbf{r} \tag{4.1}$$

According to the optimization theory, this problem has a unique solution if $U_i(x_i)$ are strictly concave functions. The dynamic problem is to solve this optimization problem in a distributed way. This requires the design of a source algorithm that adapts the transmission rate x_i in response to congestion feedback signals q_i, i. e., a source rate update law $x_i = f(q_i)$. Furthermore, a link algorithm must compute the price $p_j(y_j)$ based on the aggregate link rate y_j.

Kelly presents two optimal algorithms: The *primal algorithm* realizes a gradient controller. It consists of a first order source update law for $\dot{x}_i$ with a constant increase and a decrease proportional to the aggregate price, and a static penalty function for p_j, which enforce the link capacity constraints. The primal algorithm assumes that the route price is conveyed back to the source. It broadly corresponds to congestion control mechanisms where noisy feedback from the network is averaged at endsystems, which use increase and decrease rules generalizing those of the TCP congestion control [113]. The *dual algorithm* consists of a static source update law for x_i and a first order dynamic link price update $\dot{p}_j$. The price function $p_j(y_j)$ can intuitively be interpreted as follows: If the link arrival rate is greater than the link capacity, the link price increases, and it decreases if the arrival rate is less than the link capacity. Dual algorithms broadly correspond to congestion control mechanisms where averaging at resources precedes the feedback of more explicit information to the endpoints [113]. For both the primal and the dual control laws, the unique equilibrium can be obtained, and their globally asymptotic stability has been proved. Kelly also shows that the optimal control algorithms achieve *proportional fairness* in bandwidth allocation if the utility functions are logarithmic [111].

4.1.2 Congestion control requirements and design space

4.1.2.1 Fundamental requirements

In general, congestion control algorithms have to satisfy the following major requirements:

1. *Efficiency*: The utilization of the available network resources should be high.
2. *Responsiveness*: The algorithm should respond promptly to changes in the congestion conditions and transient events such as route changes or mobility events. The convergence time to reach the operating point should be small.
3. *Avoidance of heavy congestion and synchronization*: Dropping many packets during congestion events should be avoided. Congestion events may last longer than one RTT.
4. *Network independence*: The protocol should work well regardless of network characteristics, such as router buffer sizes, queue management strategies, or the path MTU. Packet networks encompass a large variety of heterogeneous networks that are realized by a multitude of technologies, which result in a tremendous variety of link and path characteristics: The link capacity can be either scarce in very slow speed radio links (several kbps), or there may be an abundant supply in high-speed optical links (several gigabit per second). Concerning latency, scenarios range from local interconnects (much less than a millisecond) to certain wireless and satellite links with very large latencies (up to a second). As a consequence, both the available bandwidth and the end-to-end delay may vary over many orders of magnitude, and they can be subject to substantial changes within short time frames. Congestion control mechanisms must also be able to deal with asymmetric routing, i. e., situations in which the forward path and the reverse path are different and potentially both congested.
5. *Application independence*: Congestion control has to deal with quite diverse application sending behaviors. The amount of data that an application may send varies over many orders of magnitude, and the arrival pattern may be arbitrary.
6. *Robustness and stability*: The mechanisms should be robust against noise in the congestion signals. Oscillations should be avoided. Congestion control can be viewed as a classic negative-feedback control problem with delayed feedback signals. Congestion control aims at asymptotic stability, i. e., it should converge to a certain state irrespective of the initial state of the network.
7. *Scalability*: The mechanism must work in a global network that interconnects potentially billions of endsystems. This requires *decentralization*. With the currently available technology it is impossible to realize a centralized, per-flow resource management on global scale, even if the corresponding business and legal aspects would be solved.
8. *Simplicity*: The implementation complexity and the amount of state in endsystems should be moderate. Per-flow state in core network components should be avoided, since it can hardly be realized with existing technology. Simple solutions are also more likely to become a widely accepted standard.
9. *Ability to deal with uncooperative entities*: Any solution in a multi-domain environment must consider potentially untrusted or malicious sources, sinks, and network entities on the path, as well as outside attackers.

4.1.2.2 Fairness

A further requirement for resource sharing is *fairness*. A definition of fairness is non-trivial, since it affects both technical and economic aspects. Numerous fairness concepts have been proposed [246]. For bandwidth allocations, there are three formal definitions for the fairness of a feasible allocation of data rates $\mathbf{x}$ among different connections:

- *Max-min fairness*: An allocation $\mathbf{x}$ of dimension n is max-min fair if and only if an increase of any rate must be at the cost of a decrease of some already smaller rate. Formally, for any other feasible allocation $\mathbf{x}'$, if $x'_i > x_i$ there must exist some j such that $x_j \leq x_i$ and $x'_j < x'_j$. Depending on the network topology and the traffic matrix, a max-min fair allocation may or may not exist. If it exists, the solution is unique.
- *Proportional fairness*: An allocation $\mathbf{x}$ is proportionally fair if and only if $\sum_{i=1}^{n} \frac{x'_i - x_i}{x_i} \leq 0$ for any other feasible allocation $\mathbf{x}'$. According to this fairness definition, which originates from game theory, any change in the allocation must have a negative average change. In a given setup there exists one unique proportionally fair allocation.
- *Utility fairness* or *cost fairness*: A utility fair allocation $\mathbf{x}$ maximizes a cost function that depends on utility functions. Proportional fairness is one special case of utility fairness with $U_i(x_i) = w_i \ln(x_i)$.

The most well-known metric to quantify fairness is Jain's fairness index [99]:

$$FI = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n\left(\sum_{i=1}^{n} x_i^2\right)} \in (0,1] \tag{4.2}$$

An allocation with equal values is characterized by $FI = 1$, whereas a totally unequal allocation has a fairness index of $FI = 1/n$.

In the Internet, two further fairness notions exist. The term *TCP compatibility* is defined as follows: "A TCP-compatible flow is responsive to congestion notification, and in steady state it uses no more bandwidth than a conforming TCP running under comparable conditions" [RFC 2309]. Furthermore, a flow is considered *TCP friendly* [RFC 5348] if its throughput is less or equal than the rate of a long-lived TCP connection, which is approximately given by the Equation (4.7) that is introduced later in Section 4.2.1.3. Both definitions are not rigorous, and the terms are also not consistently used. Due to the design of the standard TCP algorithms, which are discussed in Section 4.2.1.3, *TCP compatibility* inherently implies *RTT unfairness*, i. e., connections with a shorter RTT will in average obtain a higher share of bottleneck bandwidth. Also, there can be unfairness with respect to packet sizes [172].

The *TCP compatibility* is subject to ongoing debates in research and standardization communities. It is accepted that in high-speed networks new congestion control algorithms may be moderately more aggressive than standard TCP. But there is no consensus whether RTT fairness is a desirable design goal. There is also disagreement about the right *granularity* of fairness. The common goal of widely deployed mechanisms is equal bandwidth allocation among flows. However, this *flow rate fairness* is biased towards users that use many parallel flows. Briscoe [33] argues in favor of *cost fairness*, which takes into account the amount of congestion caused by a user. Floyd *et al.* [RFC 5290] disagree and state that some form of rough flow rate fairness is an appropriate goal for simple best-effort traffic.

Fairness is also an economic issue. Most congestion control schemes require the involved parties to behave in a cooperative way. However, for an individual user it is not necessarily the

Table 4.1: Congestion control design space. The design choices of TCP are highlighted. Explicit network feedback is addressed separately in Table 4.2.

Design criteria	**Degrees or freedom**
Location	*Sender* vs. receiver vs. other entity
Protocol layer	*Transport layer* vs. link/network/shim/application layer
Responsiveness	*Reactive* vs. proactive vs. no congestion control
Network feedback	*Implicit* vs. explicit
Congestion detection	*Loss-based* vs. delay-based vs. hybrid
Playout control	*Window-based* vs. rate-based
Granularity	*Per connection* vs. per aggregate
Precedence	*Best effort* vs. relative prioritization
Fairness	*Inter-protocol* ("TCP friendly") vs. intra-protocol vs. no fairness
	Flow fairness vs. utility fairness

optimal strategy to reduce the sending rate upon detection of congestion. If users act in a selfish manner and try to improve their own position by using more resources, this will result in a *tragedy of commons* problem. In economics, there are three fundamental approaches to deal with *congestion externalities*: Social norms, rationing, and pricing. The first two reflect the current situation in the Internet, where most endsystems use congestion control and there are some network fairness enforcement mechanisms such as fair queueing and scheduling (cf. Section 4.2.2.2). The latter approach would require *congestion-based charging*, which could theoretically be a basis for congestion control. However, due to the unpredictability of such mechanisms there are serious reservations both by customers and by network operators [186].

4.1.2.3 Design space of congestion control methods

Congestion control can be realized by a multitude of methods. [RFC 2914] states that "any form of congestion control that successfully avoids a high sending rate in the presence of a high packet drop rate should be sufficient to avoid congestion collapse". Table 4.1 provides an overview of the large design space.

Figure 4.3 classifies the congestion control mechanisms that are further investigated in this work. The most fundamental design choice is whether congestion control is realized *end-to-end*, i. e., solely by the sender and/or the receiver, or *network-supported*. This difference also roughly corresponds to Kelly's distinction between primal and dual algorithms (cf. Section 4.1.1.4). In an end-to-end scheme, the endsystems continuously monitor the path and increase or decrease the sending rate based on *implicit assumptions* about the path. Network support is realized by *explicit signaling* from network components that provide feedback concerning the congestion on the path, or derived metrics, such as prices. The differences and implications of these two designs are evaluated throughout the following sections.

4.1.3 Classification of end-to-end congestion control methods

Without network support, endsystems can detect congestion only by two signals: Packet loss and/or delay. Loss-based congestion control interprets lost packets as sign for congestion and reacts by reducing the sending rate. This corresponds to a binary feedback model. A fundamental drawback is that packet losses should be rare events and therefore provide a coarse

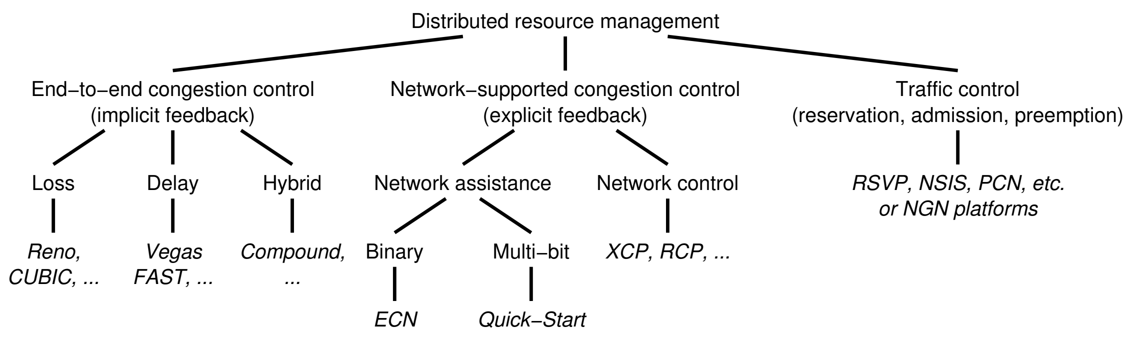


Figure 4.3: Overview of different resource management principles and examples

information only. Loss-based congestion control saturates network buffers unless active queue management schemes are used, which are introduced in Section 4.2.2.2. An alternative mechanism are delay-based congestion control algorithms that use the delay as primary congestion signal. They determine the minimum RTT and interpret increasing delays as a congestion signal. Delay can be measured more frequently and with a finer granularity than loss. Delay-based schemes can detect incipient congestion before buffers overflow. But they have to cope with two problems: The noise in packet delays has to be filtered out, and it is inherently difficult to distinguish between full and empty queues. As a consequence, realistic delay-based algorithms require a loss-based component, too.

The most widely used class of congestion control mechanism is sender-oriented, end-to-end, and loss-based. This class controls the sending rate by modification of a *Congestion Window* (CWND). A control law increases the Congestion Window if there is no sign of congestion, and it decreases it when packet loss is detected. Chiu and Jain [48] developed a fundamental set of algorithms that manipulate a CWND W as a reaction to the binary feedback:

$$\text{Increase per packet in absence of congestion:} \quad W \leftarrow W + \alpha_{\text{incr}} W^i \tag{4.3}$$

$$\text{Decrease on detection of congestion:} \quad W \leftarrow W - \beta_{\text{decr}} W^j \tag{4.4}$$

The most important class of algorithm uses $i = -1$ and $j = 1$, which corresponds to *Additive Increase Multiplicative Decrease* (AIMD). The term additive is used since the CWND is increased by the additive term α_{incr} after one RTT. The resulting evolution of the CWND over time is depicted in Figure 4.4. A fundamental property of AIMD is that it converges to the optimum point of equal sharing if several competing flows share a bottleneck [48]. This characteristic can be proved by regarding the system transitions as a trajectory through a vector space, which is illustrated in Figure 4.5 for the case of two flows. Obviously, there are also several alternatives to the AIMD control law [48], such as *Multiplicative Increase Multiplicative Decrease* with $i = 0$ and $j = 1$. These other decision-making functions may not converge to equal sharing in drop-tail networks [48].

4.1.4 Classification of network-supported congestion control methods

4.1.4.1 Terminology

Network components can be involved in congestion control in two ways: First, they can implicitly optimize their functions in order to support the operation of an end-to-end congestion control, e. g., by queue management and scheduling strategies, as introduced later in Section 4.2.2.2. Second, network components can participate in congestion control via *explicit*

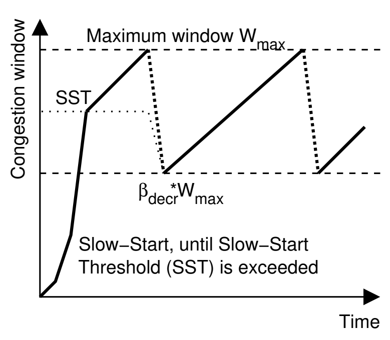


Figure 4.4: Simplified Congestion Window evolution of an AIMD algorithm

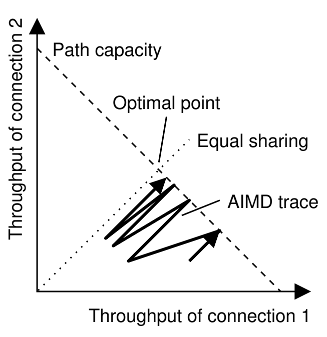


Figure 4.5: Chiu/Jain vector diagram showing AIMD's convergence to fairness

signaling mechanisms. A wide variety of terms are used to describe congestion control with explicit feedback from network components, including terms such as "router-assisted", "router-supported", "router-aided", "explicit" congestion control, etc. They are not consistently used, and the term "router" is misleading since some schemes do actually require support in all queues along a path, which may also exist in link layer devices. The author suggests to precisely distinguish between the following three terms:

Definition 4.1 (Network-supported congestion control): Network-supported congestion control schemes use explicit feedback from network components to the source of a flow. The feedback signals state of congestion.

Definition 4.2 (Network-assisted congestion control): This class of network-supported congestion control mechanism leaves the decision on sending rates to the sources. It can operate even if the sources use another congestion control mechanism.

Definition 4.3 (Network-controlled congestion control): In this class of network-supported congestion control, the network components control the sending rate of flows with a fine granularity. Sources are merely responsible for executing the control decisions.

These definitions use the term "network component" instead of "router". All network-supported congestion control schemes require a communication between network components and endsystems. Since interconnection in the Internet is realized at the IP layer, signals can only be transported within the IP layer or in higher protocol layers. Only network components that process IP packets can trigger such notifications. The following sections distinguish clearly between the terms "network component" and "router"; the term "router" is used whenever the processing of IP packets is explicitly required. One fundamental challenge of network-supported congestion control is that typically not all network components along a path are routers [172].

The focus of network-supported congestion control is the improvement of the resource sharing in networks that offer mainly a best effort service. Unlike network QoS mechanisms, network-supported congestion control mechanisms are lightweight. They do not provide guarantees, but they also do not require per-flow state in network components.

Table 4.2: Design space of network-supported congestion control

Design criteria	Degrees of freedom
Signaling	In-band vs. out-of-band
Indication extent	Single metric vs. multiple metrics
Notification extent	Single metric vs. multiple metrics
Ind./notif. transport	IP header vs. shim layer vs. transport protocol header
Feedback direction	End-to-end echoing vs. direct return vs. hop-by-hop ("back pressure")
Feedback transport	Shim layer vs. transport protocol header vs. separate protocol
Returned metric(s)	Absolute rate vs. relative rate change vs. derived metrics (e. g., price, congestion probability)
Granularity	Binary vs. multi-bit vs. discrete steps vs. fine grained
Frequency	Per packet vs. about once per RTT vs. event-triggered
Required support	Some routers vs. all routers vs. all queues vs. off-path entities
Resource management	Bandwidth pooling vs. oversubscription

4.1.4.2 Design space

Congestion control with explicit signaling has various advantages: Due to the explicit feedback, connection endpoints can obtain accurate, fine-grained information about the current network characteristics on the path. The buildup of queues and congestion situations can be detected faster compared to any implicit, inference-based method. Network components can share local knowledge about the available link capacity, which cannot easily be determined by any end-to-end congestion control. Also, endsystems cannot rapidly detect whether an unknown path is already congested. In order to be stable, any end-to-end congestion control must use a conservative rate increase and an aggressive window decrement policy [250], which is inefficient in environments with a large BDP. Network support thus allows sources to make more informed decisions, which can improve the performance, reduce the probability of packet loss, and help to improve fairness among different flows. However, there are also drawbacks, which are addressed in Section 4.7.1.2.

There are several degrees of freedom concerning the involvement of network components in congestion control. This design space is surveyed in Table 4.2. Any explicit feedback requires signaling. As explained in Section 2.1.1, signaling can be realized either by *in-band signaling* or by *out-of-band signaling*. The latter case requires additional protocols and a secure binding between the signals and the packets they refer to. This is why most network-supported congestion control mechanisms use in-band signaling. The protocol mechanisms can be implemented at different protocol layers, in particular at the network layer and/or the transport layer. In both cases, additional information can be transported by header options. A further alternative is to introduce an additional intermediate *shim layer* between network and transport layer.

Network-supported congestion control protocols use three different types of information: The *indication* is information provided by the sender about the traffic flow and its own capabilities. Network elements provide information about their state in form of *notifications*. The information that is finally provided to the sender is called *feedback*. There are also three different possibilities how the feedback can be transported to the sender: Most proposed schemes use *end-to-end echoing* as illustrated in Figure 4.6. In this case, the sink sends the feedback back to the source, either by piggybacking it on transport protocol messages, or by a shim layer protocol. The feedback reflects the notification information accumulated over the path. Alternatively,

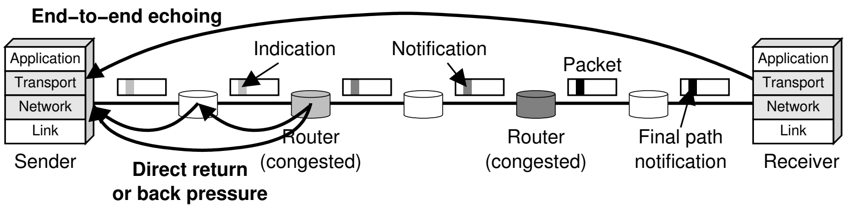


Figure 4.6: Different realization alternatives for explicit feedback from the network

network elements could directly send messages back to the sender (*direct return*), or use a hob-by-hop reverse channel if it is available (*back pressure*). However, the latter two mechanisms have several disadvantages compared to end-to-end echoing:

- New messages are created on a congested path and may increase reverse path congestion.
- The information transported in the messages must be secured against packet loss.
- A security mechanism is required so that the source can verify that the messages indeed originate from a network element on the path. Otherwise, third parties could attack arbitrary connections by forging feedback, either causing them unnecessarily to slow down, or to increase their rate, which could be used as a *Denial-of-Service* attack [211].

Another important characteristic is the *expressiveness* of the network feedback, which refers to the amount of information about the state of the network that is returned to the source. The expressiveness depends on the *granularity* and *frequency* of the feedback. For example, binary feedback schemes have a rather low level of expressiveness.

4.1.4.3 Network support and the end-to-end argument

Network support could be considered as a violation of the end-to-end argument [188] and the fate-sharing principle [50], which are explained in Section 2.2.2. However, this is not necessarily true. Congestion control cannot be realized as a pure end-to-end function only [157]. Congestion is an inherent network phenomenon and can only be resolved efficiently by some cooperation of endsystems and the network. Congestion control in today's Internet protocols follows the end-to-end design principle insofar as only minimal feedback from the network is used. The endsystems only decide how to react and how to avoid congestion. This suggests that network assistance does not violate the end-to-end argument and the fate-sharing principle, as long as per-flow behavior inside the network is avoided. In contrast, it is less clear whether network-controlled congestion control is indeed compatible with the end-to-end argument.

4.1.5 Differences to related mechanisms

4.1.5.1 Congestion control vs. bandwidth estimation

There are two ways to estimate the capacity or available bandwidth of a path: *Passive measurements* monitor ongoing data transport. This is what congestion control mechanisms do. *Active measurements* inject additional probe traffic into the network and observe the response of the network. There are different active measurement techniques [179]:

- *Variable packet size probing* estimates the capacity of individual hops by measuring the RTT from the source to each hop on the path as a function of the probing packet size, using for example ICMP error messages. The capacity is successively determined from the relation of the minimum RTTs and the probe packet sizes.

- *Packet dispersion* methods estimate the end-to-end capacity. The basic mechanism is *packet pair probing*, which sends two packets back-to-back. After traversing the narrow link, the time dispersion between the packets is linearly related to the narrow link capacity. *Packet train probing* uses multiple back-to-back packets. If tight and narrow link are identical, it is also possible to estimate the available bandwidth (*probe gap model*).
- *Self-loading probing* (also known as *probe rate model*) uses self-induced congestion. The available bandwidth is estimated by increasing the load on the path in several iterations. Increasing delays indicate that the available bandwidth is exceeded. Two important methods are *self-loading periodic streams* and *trains of packet pairs*.

There are also further methods to measure the available bandwidth, in particular the trivial mechanism to measure the data rate of a bulk data probing transfer. If the path capacity is explicitly known, fast *direct probing* methods can obtain the available bandwidth without iterative probing. However, there is no currently known technique to measure the available bandwidth of individual hops [179]. Many bandwidth estimation tools implement the algorithms presented here, which are surveyed for example in reference [179].

A fundamental shortcoming of bandwidth estimation methods is that they have to trade off *precision*, *speed*, and *intrusiveness*. There is no known method to estimate the capacity or available bandwidth with high accuracy, fast convergence, and with minimum intrusiveness [101, 214, 2]. Dispersion-based techniques are fast, but not very accurate, while self-loading probing is very intrusive. The measurement duration of iterative methods is of the order of several dozens of seconds. All bandwidth estimation methods assume that the path characteristics and average rate of the cross-traffic remain almost constant during the measurement period.

Active bandwidth estimation tools are used for a variety of reasons, such as SLA verification, fault detection, and the topology management of overlay networks. But as congestion control requires detection times within few RTTs and minimal intrusiveness, active bandwidth estimation is not considered to be reliable enough for congestion control purposes [246].

4.1.5.2 Congestion control vs. admission control

Traffic management can be either realized by *proactive* or *reactive* control [115]. Congestion control is a reactive, *closed-loop* scheme that is applicable if traffic flows can adapt their rate. Proactive or preventive schemes use admission control. Unlike congestion control, proactive schemes can provide QoS guarantees if certain constraints are fulfilled (cf. Section 2.1.4), and they do not require permanent feedback (*open-loop*). Admission control can be realized with a centralized Policy Decision Function (PDF), a *bandwidth broker*, or a distributed solution.

There are many degrees of freedom how to design admission algorithms [249]. *Parameter-based admission control* is based on worst case bounds derived from the traffic parameters that are *a priori* provided by the source, such as average and peak bit rate. Stochastic mathematical models can be used to estimate the *effective bandwidth* of the traffic. However, it is inherently difficult to describe the self-similar, long-range dependent and time-varying traffic in packet networks appropriately. Furthermore, parameter-based admission control algorithms require the deployment of traffic conditioning mechanisms such as token bucket shapers.

The other solution is *Measurement-Based Admission Control* (MBAC) [29]. Network components performing MBAC periodically estimate the available bandwidth by measurements or by probing. Admission decisions are based on a worst-case traffic descriptor of the new flow, such as the peak rate. Because MBAC scheme base admission control decision on the existing traffic instead of worst-case bounds, they can achieve a higher link utilization. There is a plethora of MBAC algorithms. However, due to the difficulty to predict future traffic, it has been found that simple and complex MBAC algorithms have similar performance and that most schemes do not meet statistical QoS targets [29], or they require excessive parameter tuning [78]. There are also hybrid solutions between parameter-based and measurement-based admission control that circumvent some problems by learning from the past experience [154].

A further concept is *endpoint admission control* [30]. In this case, admission control is entirely performed by the endsystems without network support. Before starting a new flow, the source probes the available bandwidth by one of the methods discussed in the previous section. It then decides whether to admit the new flow based on statistics of probe traffic, e. g., the number of lost packets. A fundamental drawback of endpoint admission control is that the probing is intrusive and may result in rather long setup delay. This technique is hardly suitable for applications where humans are waiting. Furthermore, flash crowd arrivals of flows can cause the system into a *thrashing* regime, i. e., the cumulative probe traffic prevents further admissions.

Admission control and congestion control mechanisms are not mutually exclusive and can be combined, as discussed in Section 4.5.2.3. Yet, an important difference is that admission control policies deny access to a new flow, whereas congestion control mandates throttling for existing flows. It is a very fundamental architectural question whether a network should deny access to flows or not [212].

4.2 State-of-the-art and open issues of Internet congestion control

4.2.1 Internet standard solution

4.2.1.1 Principles and assumptions

The three traditional cornerstones of Internet control are peering and policy configuration (on long timescales), routing path selection and traffic engineering (on medium timescales) and congestion control (on short timescales). Congestion control is an indispensable mechanism for maintaining the stability and avoiding a congestion collapse (cf. Section 2.2.1). The resource allocation in the Internet is based on a number of fundamental architectural principles:

- *End-to-end congestion control in transport protocols*: The transport protocol instances in endsystems are responsible for sensing congestion and reducing their sending rate when appropriate. This design is aligned with the end-to-end principle (cf. Section 2.2.2).
- *Trust in endsystems*: It is assumed that most endsystems voluntarily respond to congestion and that all important applications are adaptive.
- *Round-Trip Time as fundamental time constant*: The control algorithms react with a granularity of the order of the path RTT, which is the minimum feedback delay.
- *Sufficient amount of buffering*: There is enough buffering in network entities so that a congestion control algorithm can operate with an RTT of control latency.

These Internet congestion control principles have evolved over time. Historically, it was proposed to use ICMP *Source Quench* messages [RFC 896] to throttle sources in case of network overload. This optional extension of the congestion control was based on the *direct return* principle (see also Section 4.5.4). However, the end-to-end congestion control of Jacobson [98] proved to be superior. Further improvements of the original algorithm lead to the so-called *TCP Reno congestion control*, which is the current standard in the Internet [RFC 2581].

4.2.1.2 TCP Reno

Jacobson's ground-breaking control algorithms [98] increase the send rate until congestion is detected by packet loss, and then reduce the rate. The objective is to achieve an isarithmic equilibrium [53] in which the number of packets in the path is approximately constant. This *conservation of packets* principle can easily be realized by a sliding window.

The original TCP standard only specified flow control mechanisms. [RFC 793] mandates the sender to use a sliding window mechanism with a maximum size given by the *Receive Window* (RWND), i. e., the most recently *advertised receive window*. Jacobson introduced a second *Congestion Window* (CWND), which is an estimation of how much data can be outstanding in the network without packets being lost. A TCP sender can transmit up to the minimum of the CWND and RWND. The control algorithm published in [98] distinguished between two different phases: *Slow-Start* (SS) and *Congestion Avoidance* (CA). Later, Jacobson proposed an improved algorithm that became known as *TCP Reno* and that is standardized in [RFC 2581]. It distinguishes between four different phases that are partly illustrated in Figure 4.4:

- *Slow-Start*: At the beginning of a transmission into a network with unknown conditions, the Slow-Start algorithm is used to probe the network and to determine the available bandwidth. After the connection setup, the size of Congestion Window W is set to the *initial window* w. In Slow-Start, the sender may increment W by at most MSS bytes for each received ACK that acknowledges new data. The Slow-Start ends when W reaches or exceeds the *Slow-Start Threshold* (SST), or when congestion is observed.
- *Congestion Avoidance*: When W is equal or larger than the SST, W is incremented by one full-sized segment per RTT. This phase continues until congestion is detected.
- *Fast Retransmit and Fast Recovery*: The sender can guess that a packet has been lost when there are duplicate acknowledgments. By default, the arrival of three duplicate ACKs triggers a Fast Retransmit. Then, the SST is set to approximately half of the *flightsize*, i. e., the amount of outstanding data. W is set to the same value plus three MSS. After the Fast Retransmit follows the Fast Recovery phase until the loss recovery ends.
- *Retransmission Timeout* (RTO): If the RTO expires, the SST is also set to approximately half of the *flightsize*. W is set to one segment, and the sender continues in Slow-Start.

These algorithms continuously probe the available bandwidth and correspond to an AIMD congestion control with $\alpha_{\text{incr}} = 1$ and $\beta_{\text{decr}} = 1/2$. According to Jacobson, the design rationale of $\beta_{\text{decr}} = 1/2$ is that the sender falls back to a window that worked previously [98].

The Slow-Start heuristic is of particular importance for this thesis. The original idea can be attributed to Jain [99], who suggested a linear window increase. Jacobson chose an exponential increase, since this function opens the window "quickly enough to have a negligible effect on performance, even on links with a large bandwidth-delay product" [98]. The Slow-Start has two important roles: On the one hand, it has to find an appropriate sending rate for a network path that is unknown, for instance, when the connection is set up. The algorithm probes the available bandwidth of the path, and it guarantees that the source sends data at a rate that is at most twice as large as the maximum possible rate on the path. On the other hand, it must also start the *self-clocking mechanism*. In a window-based protocol, the transmission of new packets

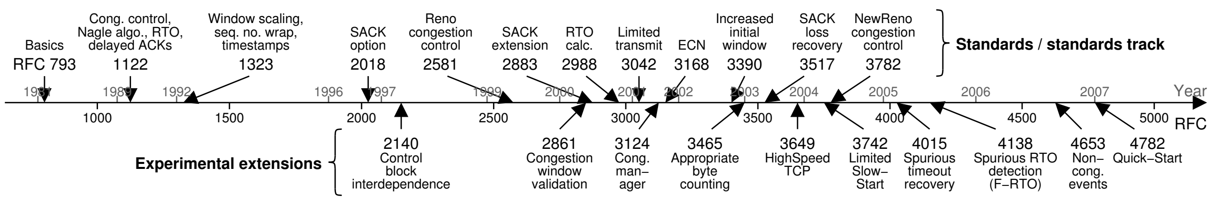

Basics
RFC 793
Cong. control, Nagle algo., RTO, delayed ACKs
1122
Window scaling, seq. no. wrap, timestamps
1323
SACK option
2018
Reno congestion control
2581
SACK extension
2883
RTO calc.
2988
Limited transmit
3042
ECN
3168
Increased initial window
3390
SACK loss recovery
3517
NewReno congestion control
3782
Standards / standards track
Year
RFC
2140 Control block interdependence
2861 Congestion window validation
3124 Cong. manager
3465 Appropriate byte counting
3649 HighSpeed TCP
3742 Limited Slow-Start
4015 Spurious timeout recovery
4138 Spurious RTO detection (F-RTO)
4653 Non-cong. events
4782 Quick-Start
Experimental extensions

Figure 4.7: IETF specifications related to TCP's congestion control, covering both the standards track documents as well as selected experimental extensions

is controlled by the stream of received ACKs. When there are no packets in the network, this process needs bootstrapping in order to limit the burstiness of the sent traffic.

If a sender has been idle for a relatively long period of time, new segments cannot be clocked out by arriving ACKs. If the Congestion Window remained unchanged, a source could potentially send a burst of the size of the CWND with full line rate. In order to prevent such bursts, [RFC 2581] recommends to reset the Congestion Window to the *restart window* if TCP has not sent data in an interval exceeding the retransmission timeout, i. e., it starts the transmission again in the Slow-Start mode. The *restart window* is equal to the *initial window*. An experimental extension [RFC 2861] describes an alternative *Congestion Window Validation* and suggests to decay CWND roughly by factor two once per duration of the RTO, while using the SST to save information about the previous value of CWND. [RFC 2861] also provides recommendations for application-limited periods, i. e., when an application sends less data than the CWND allows.

Originally, the *initial window* was one MSS. Today, [RFC 3390] permits an initial window of

$$\min(4L, \max(2L, 4380\,\mathrm{B})), \tag{4.5}$$

which depends on the MSS L and corresponds to $w_{\max} = 3$ segments for $MTU = 1500\,\mathrm{B}$, which is the default MTU value in Ethernet and supported by most Internet paths.

This section can only give a brief overview of the TCP Reno algorithms, and it does not cover all subtle aspects. Even the specification [RFC 2581] leaves open several details. For instance, it does not specify an initial value for SST, which may be arbitrarily high. Congestion control issues are also discussed in many other IETF documents. Figure 4.7 lists the important IETF documents that affect the TCP congestion control. A complete survey, including transport protocols other than TCP, has been compiled by Welzl [247]. Further information about the status of TCP standardization can also be found in [RFC 4614].

4.2.1.3 *Performance of TCP Reno*

The throughput of a TCP connection is determined by many factors:

- *Path characteristics*: The TCP throughput is always limited by the available path capacity. Further network characteristics that can affect the performance include packet transmission errors, link failures, routing problems, packet reordering, or large variations of the available bandwidth and/or delay [195].
- *Sending and receiving applications*: A TCP connection can only use the available bandwidth if the sending application provides sufficient data, and if the receiving application processes the data as fast as TCP delivers it. Applications that can always send data are characterized as *greedy*; an example are bulk data file transfers. *Application limitation* refers to the case that the sending or receiving application is the bottleneck.

- *Socket buffers and flow control*: Both the send and receive buffers may not be large enough. On the sender side, the send buffer constraints the amount of unacknowledged data (*sender limitation*). In order to allow retransmissions, the send buffer must be larger than the BDP. The receive buffer limits the maximum amount of data that can be received out-of-order. Therefore, the receive buffer should be larger than the BDP, too. Even if buffer space is available, TCP's flow control mechanisms can prevent the announcement of a sufficiently large receive window, in particular, if the sink does not use the window scaling. The latter cases are also named *receiver limitation*.
- *Congestion control*: If the Congestion Window is smaller than the BDP, the source cannot fully utilize the available bandwidth.

Models for the performance impact of the Reno congestion control exist for two special cases: Short flows and long flows. In the former case, the TCP Slow-Start has a significant influence, which is studied in Section 6.4.1. Concerning bulk data transport, there are two well-known analytical models for the throughput. The older model was developed by Mathis *et al.* [147]. It predicts the TCP throughput as

$$G = \sqrt{\frac{3}{2\eta}} \cdot \frac{L}{\tau \cdot \sqrt{p}}, \tag{4.6}$$

where L is the MSS and τ the minimum RTT without queueing delays. p is the segment loss rate, and η is the number of segments acknowledged in one received ACK. This model assumes that TCP can recover from all packet losses by fast recovery. Therefore, Equation (4.6) is only valid if the loss rate is small. Because the TCP throughput is inversely proportional to the RTT and the square root of the loss rate, this model is also known as "SQRT model".

The more complex "PFTK model" is named after the initials of its inventors [168]. It takes into account retransmission timeouts and a potential limitation by a maximum window $W_{\max}$. The average send rate of a bulk data transfer is approximated by the formula

$$G = \min\left(\frac{W_{\max}}{\tau}, \frac{L}{\tau \cdot \sqrt{2\eta p/3} + f(p)}\right) \quad \text{with} \quad f(p) = T_o \min\left(1, 3\sqrt{\tfrac{3\eta p}{8}}\right) p \left(1 + 32p^2\right). \tag{4.7}$$

T_o is the duration of the retransmission timeout, which is usually dominated by its minimum value [195]. The PFTK model captures the TCP Reno behavior over a large range of loss rates.

Both models have many limitations. They do not take into account the Slow-Start and make many simplifying assumptions. Even errors in the derivation of Equation (4.7) have been identified, which result in a theoretical over-prediction of the send rate [47]. There are several model extensions that consider further effects, but they result in a more complex formula (e. g., [59]).

Low [139] shows that the TCP Reno congestion control corresponds to a utility function

$$U_i(x_i) = \frac{1}{\sqrt{2/3} \cdot d} \arctan\left(\sqrt{2/3} \cdot d \cdot x_i\right) \tag{4.8}$$

in Kelly's optimization framework, which is introduced in Section 4.1.1.4. In Equation (4.8), d is the effective RTT including the mean queueing delay.

4.2.2 Interaction with network entities

4.2.2.1 Buffer sizing

TCP's AIMD strategy is designed to fill the buffer in front of the bottleneck. Therefore, the buffer size in network components is a crucial factor. Buffer sizing is a multi-criteria optimization problem with at least three objectives: First, buffers are required in packet networks in order to absorb short-term traffic bursts. Such transient bursts are an inherent characteristic of window-based protocols. Second, buffers must be large enough to ensure that the link utilization is high, in particular for flows using an AIMD congestion control. But, third, buffers must not be too large, since they can result in persistent queueing delays and implementation costs.

The dimensioning of router or switch buffers has long been considered a "black art". Historically, the size is determined by the *bandwidth-delay product rule-of-thumb* [233]. This guideline states that the buffer size B (here, in bit) should be equal to the capacity r of the outgoing link multiplied by the RTT τ of a connection that may be bottlenecked at that link. The rule prevents throughput underflow if one TCP connection with $\beta_{\text{decr}} = 1/2$ traverses this link.

Newer research results suggest that the buffers of network interfaces can be made much smaller if the number of flows is sufficiently large. Appenzeller *et al.* [11] argue that a buffer size $r \cdot \tau / \sqrt{n}$ is sufficient to saturate a link when n independent, long lived and not synchronized TCP connections share a bottleneck. According to this model, metro and core routers with a large number of flows n need interface buffers much smaller than the worst-case BDP. BDP-sized buffers are only useful if $n \ll 100$. Newer research results argue in favor of further reducing buffer sizes and recommend buffers between 20 and 50 packets for core routers [234]. Reference [234] also comprehensively surveys other recently proposed buffer sizing strategies. Still, there is no universal design guideline for buffer sizing so far. In general, larger buffers tend to trade off a potential increase of throughput against larger delays and jitter. An optimal buffer size can, if at all, only be derived for a specific network topology, congestion control algorithm, and application workload, and it will not be optimal in other scenarios.

4.2.2.2 Queueing schemes

An alternative to FIFO drop tail buffering is *Active Queue Management* (AQM) [RFC 2309]. The combination of drop tail queueing schemes and TCP congestion control can cause two problems: First, any loss-based congestion control operates on full queues and may thereby cause significant queueing delay. And second, phase effects and lock-out phenomena can occur, i. e., a small number of flows monopolizes the queue space, resulting in significant unfairness.

AQM avoids these problems by detecting queue buildup and notifying the sources before the queue overflows, either by packet dropping or by *Explicit Congestion Notification* (ECN). The original AQM scheme is *Random Early Detection* (RED) [RFC 2309]. RED estimates the *average queue size* and decides on packet markings or drops based on a stepwise defined function. When the average queue size is between a minimum and a maximum queue size, the probability of marking or dropping varies between 0 and a maximum drop probability. Numerous enhancements of this original RED design have been developed, which are surveyed for example by Welzl [246]. A fundamental problem is the configuration, since the optimal parameter set depends on the number and characteristics of the flows. This has lead to a plethora of AQM parameter self-tuning mechanisms [246]. A further, orthogonal design question is whether to use *tail dropping* or *front dropping*. While it is intuitive that front dropping could reduce delay in some cases [252], it never got widely accepted.

One specific design target of AQM mechanisms is fairness improvement. There are two general approaches to deal with “unfriendly” flows [222]: In case of the *identification approach*, network elements detect flows, e. g., by DPI, and enforce special policies. The *allocation approach* isolates the flows and assigns bandwidth in a fair way. A straightforward mechanism is to replace the single FIFO queue my multiple *virtual queues*, which are served in a round-robin fashion. The flows are mapped to these bins by hash functions (e. g., [68]). Unresponsive flows can be detected by monitoring the bins. Another well-known AQM mechanism that punishes unresponsive flows and improves fairness simply compares an arriving packet randomly to a queued packet and drops both if they belong to the same flow [171]. There are several further mechanisms how fair queueing could be achieved without keeping per flow state [222].

There is empirical evidence that RED or other AQM can significantly improve performance of Web applications. Experimental studies revealed that AQM can reduce the response times compared to simple FIFO drop tail queues if the link load is high [127]. However, RED is still only partly used in the Internet, and router vendors often do not enable it by default.

4.2.2.3 Explicit Congestion Notification (ECN)

Explicit Congestion Notification (ECN) [RFC 3168] has been standardized to signal congestion back to senders by a one bit feedback. Instead of dropping packets, congested routers set the *congestion experienced* codepoint in the IP header. The usage of ECN in combination with TCP requires in total four bits – two in the IP header, and two in the transport protocol header. ECN uses the two bits in the IP header to encode whether the endsystems are able to understand ECN signaling and whether the packet experienced congestion. One bit in the transport protocol header echos back the congestion experienced codepoint to the sender. The other flag is used by the source to inform the sink that the CWND has been reduced, i. e., that the congestion notification has indeed been received.

The main benefit of ECN is that packets do not have to be dropped during congestion. Routers that support ECN must implement an AQM scheme, since ECN markings must be set before a buffer overflows. ECN can be incrementally deployed in the Internet, as it does not require support by *all* routers along a path. ECN is indeed implemented in several router platforms and supported by modern TCP stacks. However, it is often not enabled by default, and therefore still hardly used in practice. The main reasons are interworking problems with “blackholes” [247], i. e., routers or firewalls that are misconfigured to discard packets with the ECN-capable bits set, as well as erroneous middlebox implementations [219]. It has also been questioned whether ECN indeed results in significant benefits. For instance, a study revealed that ECN does not necessarily improve the performance of interactive applications [127].

4.2.3 Survey of experimental new algorithms and enhancements

4.2.3.1 Shortcomings of the TCP Reno algorithms

The standard TCP Reno algorithms do not scale well to networks that have a very high path capacity $r \gg 10\,\text{Mbit/s}$ and/or an RTT $\tau \gg 50\,\text{ms}$. In order to fully utilize such paths, the CWND must exceed the BDP $r \cdot \tau$, which may require window sizes of the order of thousands of segments, or even larger. In order to sustain high steady state throughputs, TCP Reno requires

Table 4.3: Classification and comparison of important TCP congestion control variants

Algorithm	Detect.	Probing / backoff	Parameters
Reno [RFC 2581]	Loss	AI / MD	$\alpha_{\text{incr}} = 1$, $\beta_{\text{decr}} = 1/2$
HS-TCP [RFC 3649]	Loss	Convex AI / MD	$\alpha_{\text{incr}} = f(W)$, $\beta_{\text{decr}} = f(W)$
Scalable TCP [114]	Loss	Multiplicative incr. / MD	$\beta_{\text{decr}} = 1/8$
H-TCP [213]	Loss	Convex AI / MD	$\alpha_{\text{incr}} = f(t,\tau)$, $\beta_{\text{decr}} = f(G, d_{\text{max}}, \tau)$
CUBIC [184]	Loss	Concave-convex AI / MD	$\alpha_{\text{incr}} = f(t)$, $\beta_{\text{decr}} = 0.2$
Westwood+ [76]	Loss	AI / bandwidth estimation	$\alpha_{\text{incr}} = 1$
Vegas [28]	Delay	Function of RTT	Update law: Incr./decr. by 1 MSS per RTT
FAST [242]	Delay	Function of RTT	Update law: $W \leftarrow f(d,\tau,W)$
Hybla [39]	Hybrid	AI / MD	$\alpha_{\text{incr}} = f(d)$, $\beta_{\text{decr}} = 1/2$, Slow-Start modif.
Compound [227]	Hybrid	AI+delay component / MD	Reno emulation with $\alpha_{\text{incr}} = 1$, $\beta_{\text{decr}} = 1/2$
Illinois [137]	Hybrid	Concave AI / MD	$\alpha_{\text{incr}} = f(d,\tau)$, $\beta_{\text{decr}} = f(d,\tau)$

Legend: AI: Additive Increase MD: Multiplicative Decrease

very low packet loss rates $p \ll 10^{-6}$ according to Equation (4.6), which is an unrealistically low value in IP networks [RFC 3649]. The reason is that the additive increase algorithm in Reno's congestion avoidance is rather slow when the BDP is large. Furthermore, a flow may not be able to ramp up fast after a transient increase of the available bandwidth. In both cases, the path may not be fully utilized.

4.2.3.2 High-speed TCP variants

Many alternatives to the Reno TCP congestion control have been proposed. The *high-speed TCP variants* modify the algorithms that calculate the CWND, in particular when it is large. The algorithms only require sender-side modifications and are thus incrementally deployable. In the following, the most important variants ("flavors") are briefly introduced. The discussion is limited to algorithms that are generally applicable and that have a known and validated implementation in a widely used network stack. Other comprehensive surveys can be found in literature [246, 134]. Further domain-specific TCP enhancements have been proposed for wireless networks [133].

The majority of proposals belongs to the class of *loss-based congestion control algorithms* like Reno, but they use a window growth function other than AIMD. Most flavors only affect the Congestion Avoidance; the Slow-start remains unaltered. In general, window growth functions can be divided into three classes according to their shapes when being plotted over time: (a) concave, (b) convex, and (c) concave-convex. The list of algorithms in Table 4.3 contains representatives of all three cases. In literature there is disagreement concerning the optimal shape. In principle, a convex growth function is needed to ramp up the congestion window to very large values. But a convex function results in a very large window increment around the point of saturation and can cause a large burst of packet losses. As a remedy, the CUBIC congestion control [184] uses a concave-convex scheme.

Many high-speed congestion control algorithms behave like TCP Reno when being used in low-speed and/or short-distance networks. Several proposals listed in Table 4.3 also use a window growth functions that depends on the elapsed time t since the last loss event. This common design pattern significantly reduces the RTT unfairness [213]. Furthermore, almost all proposals set β_{decr} to a value smaller than $1/2$, either dependent on the Congestion Window size W, the throughput G, the minimum RTT τ, or the maximum delay d_{max}.

Another class of high-speed TCP approaches uses *delay-based congestion control*. They permanently measure performance metrics such as the instantaneous RTT d, which includes potential queueing delays, and they try to anticipate congestion before buffer overflows occurs. The control algorithm increase the CWND size if the delay d is not much larger than the minimum RTT τ (*base RTT*), and they decrease the window if the delay increases. The advantage of delay-based algorithms is that delay can be measured much more frequently than packet loss, which provides a rather coarse information if the BDP is large. Furthermore, delay-based algorithms do not completely fill bottleneck buffers. These advantages motivated the development of delay-based high-speed congestion control algorithms [242]. However, delay-based algorithms suffer from some inherent weaknesses. Delay is not a reliable congestion signal, in particular if there is delay jitter due to other effects such as *Media Access Control* (MAC) or reverse path congestion. Delay-based schemes also do not interoperate well with TCP Reno: Since delay-based algorithms back off much earlier, they only get a small share of the bottleneck capacity when competing with other flows using Reno. Recently, several *hybrid congestion control* schemes have been developed, e. g., Compound TCP [227]. These hybrid schemes combine delay-based and loss-based mechanisms (*Reno emulation*).

Another class of TCP congestion control algorithms, which is out of the scope of this work, addresses low-priority background transport. The purpose of such a *less-than best effort* congestion control is to realize the *Low-Priority Data* class [RFC 4594] or "scavenger service" without any network support. If an application uses such a low extra-delay background transport, it should be able to utilize excess bandwidth on a path without significantly perturbing other TCP connections. Most known solutions use delay-based congestion control and back off much more aggressively than Reno when detecting packet loss [232, 123]. A further option is to use inline measurements of the available bandwidth [143].

4.2.3.3 Selected algorithms: CUBIC and Compound TCP

From a practical point of view, the two most important high-speed congestion control variants are CUBIC [184] and Compound [227], since these two algorithms are enabled by default in the network stacks of popular operating systems.

The CUBIC congestion control has been developed by Rhee *et al.* [184]. It is the default congestion control in Linux since kernel version 2.6.18 and has been further developed since then [82]. CUBIC increases the CWND using a third-order polynomial function of the elapsed time from the last congestion event. This results in a concave window curve until a reference point is reached, which is the old maximum window size. If the reference point is exceeded, it continues with a convex window curve. This cubic function can be observed in the upper part of Figure 4.8, which shows the CWND evolution of a single TCP connection in a simulated scenario with a single bottleneck (cf. Section 6.1.1). The lower part of the figure presents the corresponding queue length in front of the bottleneck. After a window reduction by $\beta_{\text{decr}} = 0.2$ due to packet loss, CUBIC stores the maximum window. In the following Congestion Avoidance phase, the cubic function is then set to have its plateau at this maximum window. The motivation for this concave-convex style of window adjustment is that the sender sends for some time approximately with the previously available bandwidth and is not very aggressive at this operational point, i. e., it achieves a high link utilization without risking burst packet losses.

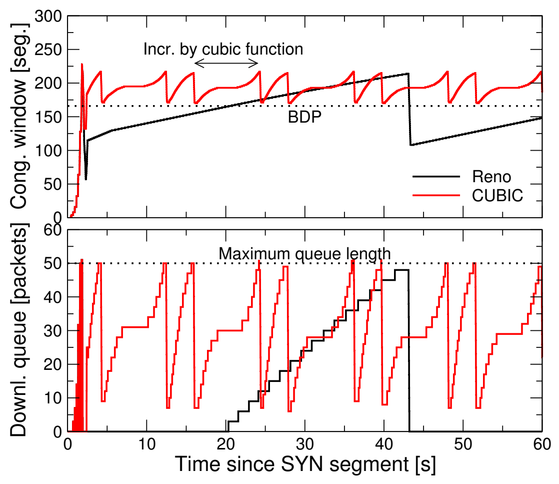


Figure 4.8: Traces of CUBIC (simulation, $r = 10\,\text{Mbit/s}$, $\tau = 200\,\text{ms}$, $B = 50$)

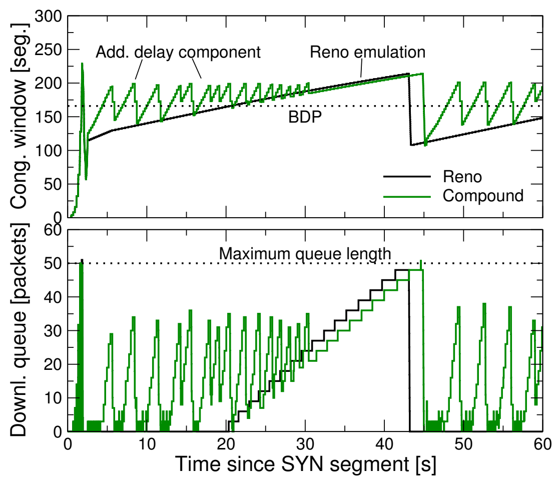


Figure 4.9: Traces of Compound (simulation, $r = 10\,\text{Mbit/s}$, $\tau = 200\,\text{ms}$, $B = 50$)

CUBIC has been excessively tested [184, 82]. It can efficiently utilize high-speed WAN paths with RTTs of 200 ms and more. Since the window growth function is independent of the RTT, CUBIC has good RTT fairness characteristics, and it behaves similar like Reno if the BDP or RTT is small. However, there are concerns about its fairness, since it has been observed that the convergence speed can be slow [130]. This is a side effect of the small multiplicative window decrease factor $\beta_{\text{decr}} = 0.2$. Other measurement results suggest that the convergence speed of CUBIC is reasonable in environments with sufficient statistical multiplexing. The *response function* of CUBIC, which is the equivalent of Equation (4.7), can be derived as follows [82]:

$$G = 1.17\,\frac{L}{\tau}\left(\frac{\tau}{p \cdot 1\,\text{s}}\right)^{3/4} \tag{4.9}$$

Another important high-speed congestion control variant is Compound TCP [227], which is enabled by default in a recent Microsoft Windows operating system. The key idea is to add a delay-based component to the standard TCP Reno congestion avoidance algorithms. The actual Congestion Window is set to the sum of a window adjusted by the Reno algorithms and a new delay-based part (*delay window*). When the network is underutilized, the delay-based part increases the window rapidly, but it falls back once queueing delays are detected. As illustrated in Figure 4.9, the CWND is lower-bounded by its loss-based component, which emulates the Reno congestion control.[1] The algorithms estimate the number of backlogged packets at the bottleneck queue and reduce the *delay window* if a threshold is exceeded, which is automatically adjusted and has a maximum value of 30 segments [227]. One can observe in Figure 4.9 that the sender reverts to Reno behavior once this limit is exceeded. Moreover, the delay-based component is only used in Congestion Avoidance when the CWND has reached a value of 41 segments. Compound TCP does not modify Reno's Slow-Start algorithms. The comparison of Figure 4.8 and Figure 4.9 shows that Compound, unlike CUBIC, results in a rather small queue length in the buffer as long as it is governed by the delay-based algorithm.

Compound TCP is a rather new algorithm, but it has undergone many tests by different research groups [227, 131]. The delay component of Compound is able to detect and effectively use

[1]The experiments in this work use an open-source patch that adds Compound TCP to the Linux kernel [9]. The original close-source implementation may behave differently.

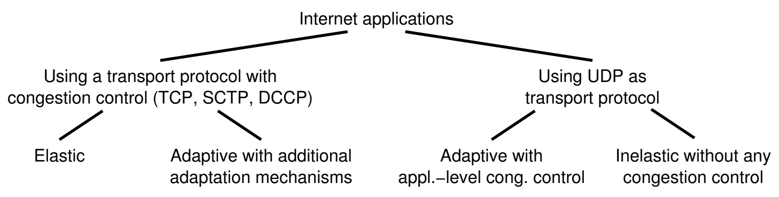

Internet applications
Using a transport protocol with congestion control (TCP, SCTP, DCCP)
Using UDP as transport protocol
Elastic
Adaptive with additional adaptation mechanisms
Adaptive with appl.–level cong. control
Inelastic without any congestion control

Figure 4.10: Classification of Internet applications with respect to congestion control usage

spare bandwidth in situations with a large capacity and/or long delay. In case of congestion, it is dominated by the loss-based component, which bounds the throughput to the value that would be achieved by TCP Reno and is thus inherently TCP-compatible. While other delay-based algorithms suffer from the problem that they only obtain a small throughput when competing against loss-based algorithms, the loss part of TCP compound ensures inter-protocol fairness. Because of these properties, it is assumed that Compound TCP can safely be deployed in the Internet. The *response function* of Compound TCP is given by the following formula [227]:

$$G = 0.225\,\frac{L}{\tau}\frac{1}{p^{0.8}} \tag{4.10}$$

4.2.3.4 Performance comparison of different algorithms

The performance implications of high-speed congestion control algorithms is subject of ongoing research work. Systematic studies [134, 80] compare several flavors under a wide range of conditions that include mixes of long and short-lived flows, different bandwidths, latencies and background traffic, as well as a range of router buffer sizes. All algorithms listed in Table 4.3 improve the utilization in a relatively static environment with long-lived flows. However, many of the proposals exhibit poor responsiveness to changing network conditions. In particular older algorithms [RFC 3649, 114] suffer from slow convergence times following the startup of a new flow and also reveal a poor intra-protocol fairness. The different studies also report RTT unfairness between competing flows with different latencies, but some results are ambiguous [134, 80]. The recently developed hybrid loss/delay-based protocols are still under evaluation, and remaining open issues have been identified [131].

4.2.3.5 Existing congestion control methods for non-TCP flows

The utilization of congestion control by the vast majority of flows is a fundamental requirement for the stability of the Internet. As TCP continues to be used by most applications, it ensures that most traffic is congestion-controlled. The alternative transport protocols SCTP and DCCP use a TCP-compatible congestion control, too. Certain applications use UDP transport instead of TCP. If these applications generate a substantial amount of traffic, they should use some kind of application-level congestion control (cf. Figure 4.10). A standardized congestion control framework for such applications is *TCP Friendly Rate Control* (TFRC) [RFC 5348]. TFRC realized a rate-based congestion control by using a simplified version of Equation (4.7).

Multimedia streaming is one type of UDP-based applications that can generate large amounts of data. The most important UDP-based audio and video streaming applications in the Internet are responsive to congestion and TCP-compatible [164, 91]. Their rate adaptation is facilitated by *scalable video codecs* [44]. As alternative, HTTP over TCP transport for multimedia content is getting more and more popular, driven by the popularity of Websites with multimedia clips.

Table 4.4: Known characteristics of popular TCP stacks

TCP mechanism	**Linux kernel (versions 2.6.18 or newer)**	**Microsoft Windows (Windows Vista or newer)**
Default congestion control	CUBIC Reno (in some distributions)	Reno (Windows Vista) Compound (Windows 2008 Server)
Window scaling (RFC 1323)	Enabled, up to 4 MiB	Enabled, up to 16 MiB
Default scaling factor	ca. 7 (depends on memory)	8 (but only 2 for HTTP)
Automatic buffer tuning	Enabled	Enabled
Delayed ACKs (RFC 2581)	Not during Slow-Start	Enabled
Initial SST value	$2,147,483,647$ B	$65,535$ B
Limited Slow-Start (RFC 3742)	Supported, disabled	Undocumented
Appr. byte counting (RFC 3465)	Supported, disabled	Enabled
Cong. Window Valid. (RFC 2861)	Enabled	Undocumented
Conn. state caching (RFC 2140)	Enabled	Undocumented
SACK (RFCs 2018, 2883, 3517)	Enabled	Enabled
Timestamps (RFC 1323)	Enabled	Supported, disabled
RTO calculation (RFC 2988)	Not compliant (min. 200 ms)	Compliant
ECN (RFC 3168)	Supported, disabled	Supported, disabled

Another use case of UDP is the high-speed transport of scientific data, e. g., in Grid computing. Several application protocols realize functions such as reliability and rate control on top of UDP. They differ in many details from TCP and in some cases they do not implement a TCP-compatible congestion control [87]. However, their usage is mostly limited to controlled environments outside the Internet. Given the advancements in TCP's congestion control mechanisms, such UDP-based protocols are only beneficial in very specific application fields.

Congestion control is not applicable to inelastic flows. An important use case of inelastic flows are *pseudowires* that tunnel constant bitrate traffic over UDP. They cannot respond to congestion in a TCP-compatible manner [36]. Then, overload situations must be handled by flow admission and termination. This requires a corresponding control plane (cf. Section 2.2.6 and Section 2.3.6), which does not exist in the global Internet. The new IETF *Pre-Congestion Notification* (PCN) architecture [RFC 5559] describes an architecture for protecting the Quality of Service of inelastic flows within a DiffServ domain. It uses the ECN-codepoints in the IP header. Within a domain, the rate of PCN traffic is metered on every links and packets are marked when certain configured rates are exceeded. Per-flow state is required at the ingress nodes of a PCN domain. The received ECN markings allow these boundary nodes to make decisions about whether to admit or terminate flows. Thus, PCN is one example of a *stateless core* admission control based on network feedback.

4.2.4 Impact of network stack implementations

Each operating system implements the TCP/IP stack in a different way, and the stacks are evolving over time [150]. As the specifications leave open many details, the TCP implementations of different operating systems differ significantly, and may also change from version to version. In general, the stacks in modern operating systems support more features and are better tuned.

There is a set of TCP enhancements that is supported by most endsystems in the current Internet. Measurements [150] show that most stacks use an error recovery with Selective Acknowledgments (SACKs). Also, the sizes of advertised receive windows have significantly increased: While a few years ago maximum windows of 16 KiB or 64 KiB were common, modern stacks support receive window scaling [189, 219]. Other proposed enhancements such as ECN get deployed only very slowly. Table 4.4 lists important TCP extensions that are supported by state-of-the-art stacks. It also illustrates some cases of different design choices. This list is not comprehensive and may change concerning newer releases of the operating systems.

As already explained in Section 3.3.4.3, this work uses the Linux networking stack, which is a powerful and highly optimized stack. Due to the availability of the source code, the Linux stack is widely used in networking research. A comprehensive, yet partly out-dated survey of the specifics of the Linux TCP implementation has been compiled by Sarolahti *et al.* [189]. A general introduction can also be found in the book of Wehrle *et al.* [240].

The Linux kernel uses the concept of congestion control modules with a common interface (cf. [241]). Since it is simple to design new congestion control modules, more than ten different congestion control algorithms are implemented in newer Linux kernels. The system configuration determines which module is used, and an application can overwrite this choice by a socket option. CUBIC is the default algorithm unless the configuration is changed. Another feature of the Linux stack is a sophisticated SACK processing engine that may even recover if retransmitted segments get lost. As shown in Table 4.4, there are also several Linux-specific mechanisms. Such an example are "QuickAcks": A Linux receiver acknowledges every segment if it assumes that the sender is in the Slow-Start phase. This disabling of the delayed acknowlegments, which does not violate [RFC 2581], can speed up the data transport in Slow-Start. A heuristic decides when to start to delay acknowlegments. The maximum number of QuickAcks is half of the advertised receive window counted in segments with an upper bound of 16. Linux also implements Congestion Window Validation [RFC 2861]. The receive window auto-tuning used in Linux and its implication are discussed in Section 5.2.1.

In addition to TCP algorithms, other mechanisms inside the operating system, including interrupt handling, locking, and process scheduling can have a significant impact on the delays of messages. Measurements have shown that delays inside the Linux operating system can be one order of magnitude larger than processing delays inside applications [238].

4.2.5 Remaining open and unsolved issues

4.2.5.1 Challenges

Despite a large amount of research, the congestion control mechanisms in the Internet face several challenges that have not been completely solved so far [172]:

- *Heterogeneity*: The congestion control algorithms have to deal with the variety and dynamics of path characteristics in the Internet, as well as with a large range of different application characteristics. Even though Jacobson's congestion control principles have proved to work well in the Internet for almost two decades, there are still many situations where today's congestion control algorithms react in a suboptimal way, resulting in low resource utilization, non-optimal congestion avoidance, or unfairness. The *flow startup* by the Slow-Start heuristic is one of the cases in which the existing standard solution is suboptimal. This open issue is addressed in the remainder of this work.
- *Stability*: The stability of congestion control algorithms have been studied extensively, but it is still unclear if TCP's algorithms are asymptotically stable under arbitrary network conditions. Also, the stability impact of Slow-Start is not entirely clear.

- *Fairness*: As discussed in Section 4.1.2.2, the Internet lacks a general concept of fairness, and there is no consensus whether TCP compatibility, RTT-fairness, flow-fairness, etc. are a design goal, or not.

In addition to these challenges, there are also numerous specific issues that include the question of network support, reaction to corruption loss, multi-domain operation, misbehaving senders and receivers, etc. These challenges are generally considered to be open research topics [172].

4.2.5.2 Alternative congestion control principles

It is always possible to tune congestion control parameters based on some knowledge about the environment, e. g., in specific wireless network technologies. In the Internet, the challenge is to define congestion control mechanisms that operate reasonably well in the whole range of existing scenarios. Some improvements of congestion control could be realized by simple changes of single functions in endsystem or network components. However, new mechanisms can also require a fundamental redesign of the overall network architecture, and they may even affect the design of applications. This can imply significant interoperability and backward compatibility challenges and/or create network accessibility obstacles. There are a number of fundamental, potentially disruptive alternatives to the current Internet congestion control principles:

Layer: Congestion control mechanisms can be placed in layer 3 or layer 4 [246]. The realization in the transport layer is a design choice of the TCP/IP protocol suite. In contrast, the OSI reference model [ISO 7498] explicitly lists congestion control as one of the functions to be provided by layer 3. These two possibilities motivated the proposal of a new, intermediate shim layer that handles congestion control (see Section 4.6).

New link-layer interfaces: The congestion control in existing transport protocols operates almost independently of lower and higher protocols. The efficiency could be improved by cross-layer signaling from link layer to higher layers (*link layer triggers*): Possible triggers include link connectivity changes (link up, link down, link bandwidth change), link-local congestion notification from link layer flow control, information about non-congestion packet loss, or other link characteristics (MTU, reordering). Such triggers would in particular be beneficial in highly dynamic environments.

New application interfaces: The sockets interface between transport protocols and applications consists of only few primitives [1003.1]. This could be extended in two directions: First, if more expressive interfaces were available, the transport layer could expose information about current network conditions to applications [61]. Second, applications could provide requirements and traffic specifications to the transport layer, which could be used to optimize the congestion control parameters and support differentiation (*tailor-made congestion control*).

No congestion control: It could be possible to build a network in which endsystems do not have to respond to congestion, and just send at their respective line rates [182]. When packet losses occur, the endsystems do not slow down sending rates. Instead, they increase their redundancy rate via erasure correction codes, such as *fountain codes*. However, such a network architecture would have numerous drawbacks: First, not using congestion control can be very inefficient in certain network topologies [246]. Since packet loss is frequent, there are many needless packet transmissions. Second, erasure encoding is not well suited for applications that only send data sporadically. And third, the realization of many transport protocol mechanisms gets more complicated. For instance, the reliable transport of control information is difficult when

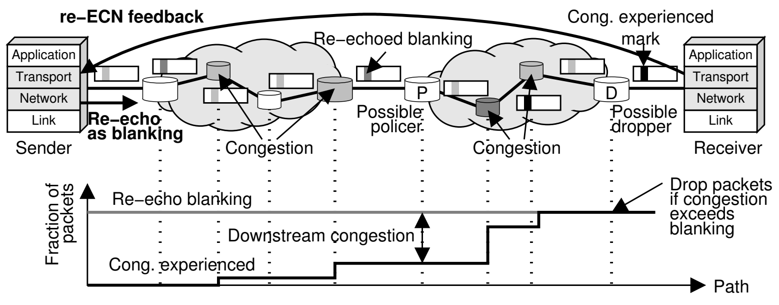


Figure 4.11: Principle operation of re-ECN: Re-echoing the fraction of packets that experienced congestion enables network-internal policing according to the downstream congestion

the packet loss probability is large [182]. There are also proposals to completely disable the TCP congestion control in dedicated circuits with a known fixed capacity [159].

Aggressive congestion control: Recently, *relentless congestion control* [146] has been proposed. Instead of using AIMD, the idea is to reduce the CWND just by the number of lost packets, i. e., to adhere strictly to the packet conservation principle. Such an algorithm is not TCP-compatible and requires fairness or traffic control mechanisms in the network in order to function appropriately in a shared environment. But the simple design facilitates such traffic management. Further research is required to study the implications of this disruptive idea. It is a general trend in the transport layer research community that packet loss is considered to be less critical.

Re-feedback: A special case of signaling is re-feedback [32]. The idea of re-feedback is to collect path information in packet header fields as packets traverse the path, return it to the sender, which then reinserts this information in the packets sent along the path. Therefore, the packets carry a view of the whole path. The re-feedback mechanism ensures that the end-to-end performance metrics are visible to the network. As described in Figure 4.11, this allows each network component on the path to predict the congestion on the remainder of the path. For instance, policers can then detect malicious endsystems not using congestion control. Re-feedback can be realized by an extension of ECN, termed re-ECN, which requires changes in the ECN code points and one additional bit in the IP header [34]. Re-ECN improves the accountability and could enable congestion-based charging.

4.2.5.3 New transport protocols

TCP, as well as other Internet transport protocols, represent only one specific realization of the large transport protocol design space [96, 211, 87]. In the Internet architecture, transport protocols combine three functional areas: Transport primitives/semantics, traffic management, and multiplexing. All these mechanisms can be realized in many different ways [95]. Table 4.5 provides an overview of these different realization options. Some alternatives are:

- *Multipath routing* and *multi-flow transport protocols* would enable multiple subflows that transport traffic among different routes [12], and also bandwidth aggregation among multiple interfaces of multihomed endsystems [93].
- *Feature negotiation* could either negotiate the use of a transport protocol during the connection setup (“meta-SYN”), or the composition of specific mechanisms within a configurable and extensible transport protocol [31].

Table 4.5: General transport protocol design space. The service features of TCP are highlighted. The design space of congestion control is surveyed separately in Table 4.1.

Design criteria	**Degrees of freedom**
Association type	*Connection oriented* vs. connection less
Delineation	*Byte steam* vs. message oriented
Encoding	*Raw transport* vs. compression vs. advanced encoding
Direction	*Bidirectional* vs. unidirectional
Ordering	*Ordered* vs. partially ordered vs. unordered
Reliable delivery	*Reliable* vs. partially reliable vs. unreliable
Aggregation	*Single stream* vs. multiple streams
Path redundancy	*Single path* vs. multiple paths
Dissemination mode	*Unicast* vs. multicast vs. anycast
Security	*No security mechanisms* vs. integrity/confidentiality protection
OS integration	*Kernel-space* vs. user-space

- *Partial reliability* and/or *partial ordering* could be useful for applications that have own reliability or ordering mechanisms [77, 116].

There have been many proposals for transport protocols that could replace TCP. Historically, there have been many non-IP transport protocols [96]. For instance, the *Xpress Transport Protocol* had modes for both window-based and rate-based transport, and sender and receiver could negotiate traffic specifications [96]. Furthermore, several new transport protocols on top of UDP have been developed [87]. There are two recent proposals for TCP replacements: Allman proposed an evolution of TCP to "TCPx2" [6]. The general idea is to double the size of the TCP header, but retain the protocol semantics. This header extension would create more space for port numbers, sequence numbers, the receive window, and also more reserved bits. This additional flexibility could enable new protocol extension in the future. Other similar proposals suggest to just extend the TCP option space. Unlike this evolutionary approach, Ford *et al.* suggested a fundamentally new transport layer design, which replaces today's TCP by three protocol layers [72]: An *endpoint layer* that realizes multiplexing, a *flow regulation layer* that offers more flexibility for realizing congestion control (hop-by-hop congestion control, multi-path transport, ...), and a *transport layer* that provides semantic abstractions and flow control. A pragmatic solution would be to use UDP as endpoint layer, because this allows immediate deployment in the Internet. Ford also proposes *structured streams* [71], which realize a simple management of streams within an association. With structured streams, applications could use one stream per transaction instead of multiple TCP connection and avoid TCP's connection establishment delays. The performance improvement of such a mechanism for Web applications has also been shown at the example of SCTP [161].

Yet, none of these proposals has been successful so far. One reason is the difficulty to build transport protocols that satisfy all requirements of all types of applications. A transport protocol combines several mechanisms, and any combination is likely not to handle well one category of traffic [54]. Another major obstacle is the widespread deployment of middleboxes in the Internet, which only know TCP and UDP and block packets with other transport protocols. Erroneous middlebox implementations even prevent the deployment of new standardized TCP features [219]. As a result, it is unrealistic that transport protocols other than TCP and UDP will be widely used in the Internet in short-term. As a consequence, the *future evolution of TCP and its protocol mechanisms* is still one of the key questions of Internet research.

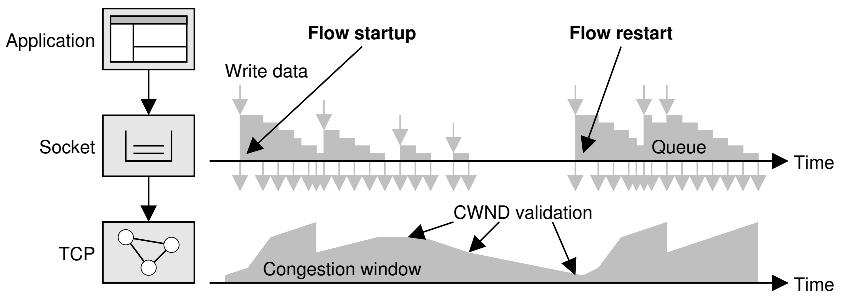


Figure 4.12: Illustration of a flow startup and a flow restart

4.3 Fast startups: Definition, motivation, and design principles

4.3.1 Definition of fast startup congestion control

The *flow startup* is one of the few cases that is not optimally solved by any existing Internet congestion control method. The flow startup situation occurs whenever a connection is established. The fundamental challenge for congestion control is that endsystems often have no information about the characteristics of the path and about the available bandwidth. As depicted in Figure 4.12, a similar *flow restart* problem also occurs after relatively long idle times, since the congestion control state then no longer reflects the current situation in the network. Both situations are not identical. A flow restart mechanism could assume that the path has not completely changed after the last data transmission, whereas a flow startup to an unknown destination does not have any information about the path.

As presented in Section 4.2.1.2, current TCP implementations use the Slow-Start mechanism both for the flow startup and the flow restart. The Slow-Start must both find an appropriate sending rate and establish the *self-clocking mechanism*. However, the Slow-Start is not optimal in many situations: First, it can take quite a long time until a sender can fully utilize the available bandwidth on a path. Second, the exponential increase may be too aggressive and cause multiple packet drops if a large congestion window is reached (*Slow-Start overshooting*). And third, the Slow-Start does not ensure that new flows converge quickly to a reasonable share of resources, in particular if they compete with long-lived flows. This convergence problem may even worsen if high-speed congestion control variants get widely used.

The flow startup is one of the remaining open issues of the Internet congestion control. The Slow-Start heuristic and its interaction with the congestion avoidance phase was largely designed by intuition [98]. In particular, one can question whether the Slow-Start should be used for restart after periods of idleness, as recommended by [RFC 2581]. This mechanism actually creates an incentive for applications to send unnecessary data during idle periods, just to keep the CWND at a large value. As the Slow-Start mechanism has been designed 20 years ago, it is more and more questioned whether it is still appropriate, since many online applications would benefit from a faster acceleration [12]. Welzl describes this trend as follows: "congestion control is about using the network as efficiently as possible. These days, networks are often overprovisioned, and the underlying question has shifted from 'how to eliminate congestion' to 'how to efficiently use all the available capacity'" [246].

Fast startup congestion control is a new design principle that addresses the latter question. More formally, the concept of fast startup congestion control is defined as follows in this work:

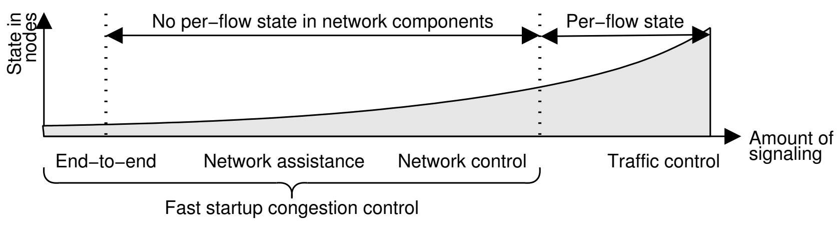


Figure 4.13: Scope of fast startup congestion control

> **Definition 4.4 (Fast startup congestion control)**: A fast startup scheme is a congestion control mechanism that permits a starting data transmission to use the available bandwidth on a network path almost instantaneously, i. e., within a time frame of the order of one or very few RTTs.

The Slow-Start algorithm fulfills this definition if the bandwidth-delay product is of the order of some MTU sizes only, but not if the BDP is larger. As a consequence, a general fast startup congestion control must be faster than the Slow-Start in networks with a large BDP, e. g., if the communication path traverses long-distance links or cellular access networks.

As illustrated in Figure 4.13, fast startup congestion control can either be realized *end-to-end*, or, alternatively, by *network-assisted* or *network-controlled* schemes, which have a higher complexity. From a conceptual point, an interesting property of the latter class of solutions is the *hybrid combination of aspects both of end-to-end congestion control and QoS traffic control.*

Substituting the Slow-Start mechanism is a non-trivial task. Any end-to-end flow startup approach has to cope with the inherent problem that endsystems lack precise information about the path characteristics when they start a new data transfer. The *root problem* is the difficulty to determine the available bandwidth. Network-supported schemes could reduce this uncertainty by querying the network components along the path, but they are currently not usable in the Internet. So far, little theory has been developed to understand the flow startup problem and its implications on congestion control stability and fairness. There is also no established methodology to evaluate whether new flow startup mechanisms are appropriate or not [172].

4.3.2 Motivation and challenges of fast startups

4.3.2.1 Overview of arguments

There are several reasons why fast startup congestion control is useful, but any solution also has drawbacks. In the following, the arguments for and against fast startups are discussed in the context of requirements of interactive applications, which are introduced in Section 3.1 and Section 3.2. The arguments are also summarized in Table 4.6. It must be noted that these arguments are valid both for the flow startup and the flow restart situation. One further argument in favor of fast restarts is that many Internet paths are rather stable on the time-scales less than a minute, i. e., one could optimistically assume that the characteristics have not changed.

4.3.2.2 Speedup

The TCP Slow-Start dominates the performance of short and mid-sized data transfers, which are very frequent in the WWW. Due to Slow-Start, their transport times mainly depend on the RTT. The *delaying effect on HTTP transfers*, which is quantified in Section 6.4, is well-known

Table 4.6: Arguments for and against fast startup congestion control

Issue	**Pros**	**Cons**
Speedup	Significant performance speedup - if RTT is large - for mid-sized data transfers	Only beneficial for selected applications
Fairness	Potentially improved fairness for - short/mid-sized flows - flows with large RTT	RTT unfairness is an inherent TCP design principle
Predictability	Startups should provide predictable transport delays	Lack of corresponding APIs to applications (sockets interface)
Congestion risk	Only limited additional risks - TCP stack efficiently handle packet loss - often per-user queues in bottlenecks - could be reduced by network support	Increases the probability of congestion - higher packet loss rate - possible negative impact on other flows

since the emergence of the WWW [88, 165]. First workarounds have been developed very early, such as reducing the content size by data compression [165]. It has also been observed early that a faster startup would improve transaction times in the Web [16]. In the meantime, the transfer sizes have increased significantly. As explained in Section 3.3.3.2, average transfer size are typically of the order of dozens of kilobyte in today's WWW. The available bandwidth in the Internet has also increased by several orders of magnitude. Still, this capacity cannot be efficiently used in the first RTTs of a data transfer. As explained in Section 3.1.4, new interactive applications have emerged that transport larger amounts of data *and* require a small latency. Given that the data rates are likely to increase further, while packet delays will not significantly decrease, TCP Reno's Slow-Start is not future-proof and a potential obstacle for new broadband interactive applications.

However, the unnecessary delays caused by Slow-Start only affect selected applications. If data transfers are very small and fit into the initial window, there is no potential for improvement. For instance, in the data presented in Figure 3.14, the size of only 1 % of the requests and of only 25 % of the responses exceeds an initial window of three MSS. From this follows that for a majority of transfers TCP's current flow startup mechanism is still optimal. But one must also note that current delay-sensitive applications already optimize their TCP connection usage in order to mitigate the Slow-Start delays, e. g., by splitting larger content into several, parallel transfers. Fast startup congestion control mechanisms could achieve the *same performance with less complexity* and thus enable simpler application designs. In general, a fast startup is useful for all responsive applications that frequently transport "mid-sized" amounts of data, i. e., neighter "mice" nor "elephants".[2] This specific range of application characteristics is also illustrated in Figure 4.14.

4.3.2.3 Fairness and predictability

The Slow-Start favors flows with a small RTT, and it does not equally assign bandwidth to short-lived and long-lived flows. Concerning RTT unfairness, it is debatable whether the flow startup should depend on the RTT, or not. *Removing the RTT unfairness* has been a goal of many of

[2]The author suggests to use the term "cheetah" for this type of data transfers.

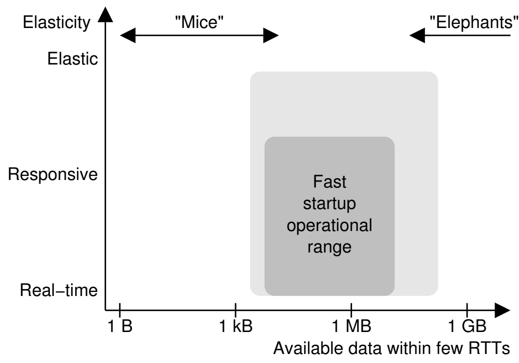


Figure 4.14: Classification of applications that benefit from fast startup schemes

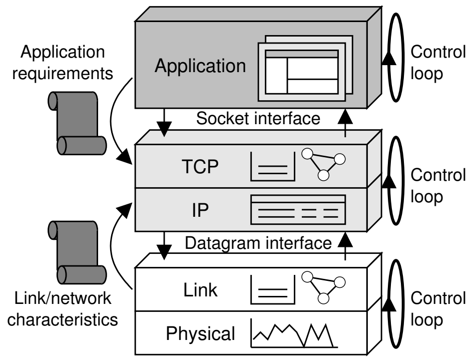


Figure 4.15: Cross-layer signaling of application and network characteristics

the new high-speed TCP algorithms for the congestion avoidance phase that are surveyed in Section 4.2.3.2. Several authors [79, 56, 35] have also pointed out that, according to scheduling theory, short flows should actually be prioritized to optimize the average completion time.

The existing flow start algorithms are completely independent of application requirements – as all TCP algorithms. An application can hardly *predict the transfer time* of a given amount of data in Slow-Start. It depends on the RTT and other parameters, even if the path has a large available bandwidth. This makes it difficult to define response time targets and SLAs as introduced in Section 3.1.3.3. The unpredictability also imposes limits on application performance differentiation and assurance mechanisms that are explained in Section 3.2.1.2. While it is impossible to guarantee deterministic delay bounds without end-to-end network QoS mechanisms, for many applications it might be useful if latency bounds could at least be met with a high probability. Fast startup congestion control could be one mechanism to fulfill such target transmission times [136]. More generally, more differentiated flow startup schemes would be one possibility to better take into account application requirements within the TCP congestion control. This requires additional application interfaces as illustrated in Figure 4.15. The syntax and semantics of these interfaces is further detailed in Section 5.1.1 and Section 7.1. Also, applications that are adaptive and have own control loops could benefit from a better control over the transmission time of data, as this could be one control parameter of application adaptation [60, 141, 26].

4.3.2.4 Congestion risk

A very fundamental design trade-off in networking is the *optimistic vs. pessimistic allocation* of resources. A small initial window pessimistically assumes that a path may always be congested. Yet, one could also optimistically assume that most paths in the Internet are right-sized and there is often a reasonable amount of available bandwidth. This could, on the one hand, reduce transport delays. But, on the other hand, it also increases the risk of additional queueing delays and packet losses. Obviously, an optimistic flow startup procedure can also have a negative impact on other flows that share the same bottleneck.

One key principle of the Internet congestion control is to *avoid useless work*, i. e., to transmit only data that can be expected to arrive at the receiver. This explains the conservative flow start behavior of the Slow-Start. However, one can question whether this design choice is still correct in an Internet with ubiquitous broadband connectivity. The risk of a congestion collapse by retransmissions, as explained in Section 4.1.1.1, is minimal, since the SACK implementations in today's TCP stacks are almost work-conserving and hardly trigger any needless retransmission. And there are further reasons why the congestion risk of a fast startup may be tolerable: In the current Internet, bottlenecks are typically located in the access networks, where per-subscriber buffers are not uncommon. In such cases, if a performance degradation of other flows would occur, it would mostly be limited to the flows of one subscriber. Additionally, buffers with AQM can absorb short bursts and enforce some level of flow fairness [136].

Finally, a network-supported flow startup scheme can reduce the risk of congestion, as it can prevent an aggressive flow start if the path is already congested.

4.3.2.5 Shortcomings of alternative application-level solutions

There are several techniques that can mitigate the performance limitations of the Slow-Start without requiring modifications of the transport protocols. However, all these techniques have drawbacks as well and cannot completely substitute a fast startup congestion control.

First, many Web browsers open *multiple parallel TCP connections* to a Web server, as explained in Section 3.1.2. When *n* parallel downloads are ongoing, each of them using a Slow-Start, the aggregated bandwidth increases *n* times as fast as a single Slow-Start. In addition to this speedup, the usage of several connections has also other advantages. As long as HTTP without pipelining is used, a single connection suffers from *Head-of-Line Blocking* (HOL), which can be avoided by concurrent requests over different connections. On a path with a large BDP, a single connection could also be limited by the TCP flow control if the endsystems do not support receive window scaling. High-speed bulk data transfers can better leverage multiple processors in an endsystem if the parallel connections are assigned to different processors. However, the use of multiple connections has numerous drawbacks, too. Each connection results in additional control traffic and connection setup overhead, and it requires memory space for the *TCP control block* in the operating system and potentially also table space in middleboxes such as NAT devices or load balancers, which becomes increasingly a scarce resource in the Internet.

The second alternative are *Performance Enhancement Proxies*. As explained in Section 3.2.2.4, a commonly used mechanism is to split end-to-end TCP connections. This splitting reduces the RTT of each individual connection and thus speeds up the Slow-Start. However, this solution violates the fate sharing principle (cf. Section 2.2.2) and causes many problems to applications that rely on end-to-end connectivity.

Third, large RTTs can be avoided by fetching content from topologically near servers. This is purpose of *Content Delivery Networks* (cf. Section 3.2.2.3). As already mentioned, CDNs are challenged by more and more dynamic content that cannot easily be cached in surrogates, so that the need for interactive downloads over paths with larger RTTs is likely to increase.

Finally, content *pre-fetching* is a technique that is increasingly used by modern Websites in order to mask the network latency and transport delays from users [209]. But this method is only successful if the prediction is indeed correct, and fails if content must be created on-the-fly.

4.3.3 Design principles of fast startup congestion control

There are numerous different possibilities how to start a flow. Figure 4.16 illustrates key design alternatives. From a protocol engineering point of view, *end-to-end mechanisms* have to

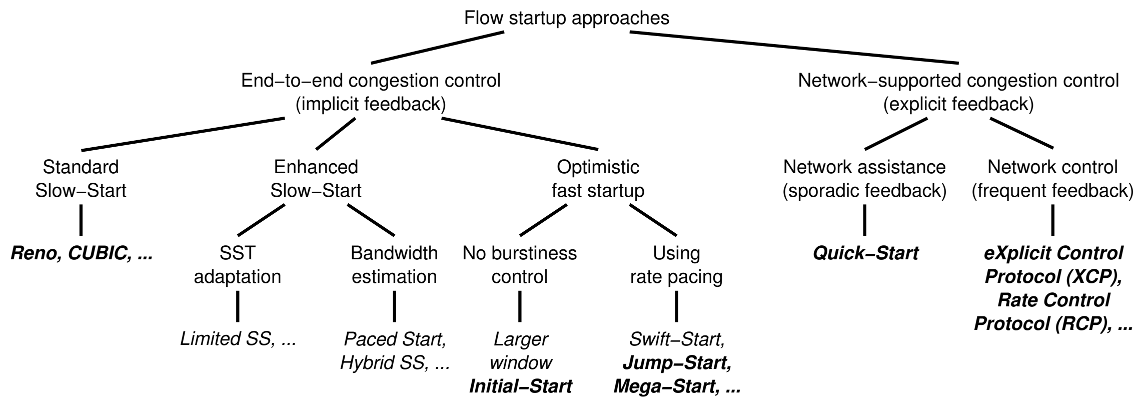

Flow startup approaches
End–to–end congestion control (implicit feedback)
Network–supported congestion control (explicit feedback)
Standard Slow–Start
Enhanced Slow–Start
Optimistic fast startup
Network assistance (sporadic feedback)
Network control (frequent feedback)
Reno, CUBIC, ...
SST adaptation
Bandwidth estimation
No burstiness control
Using rate pacing
Quick–Start
eXplicit Control Protocol (XCP), Rate Control Protocol (RCP), ...
Limited SS, ...
Paced Start, Hybrid SS, ...
Larger window
Initial–Start
Swift–Start,
Jump–Start,
Mega–Start, ...

Figure 4.16: Overview of the design space of flow startup mechanisms. The approaches that are studied in this work are highlighted.

be distinguished from *network-supported mechanisms*. There are different end-to-end enhancements of Reno's Slow-Start, which are surveyed in Section 4.4. Furthermore, several end-to-end fast startup schemes have been proposed, which are one main focus of this thesis. These approaches are further detailed in Section 4.4, too. Another comprehensive survey can be found in [RFC 4782].

End-to-end flow startup approaches cannot timely determine the available bandwidth on the path at the beginning of a data transfer. Thus, network support has two key advantages:

- *Signaling is faster than end-to-end measurements*: By signaling, a source can get information about the spare capacity within one RTT.
- *Information from the network reduces the uncertainty*: Determining the available bandwidth by bandwidth estimation is error-prone (cf. Section 4.1.5.1). While any signaled information can be wrong, too, network components can more easily determine the local load situation.

Providing congestion information to endsystems can prevent them from starting flows with a high rate, which would worsen an already existing congestion situation. Yet, signaling cannot completely solve the flow startup problem: Even if a source knew exactly the available bandwidth on a path, it would still not necessarily be safe to jump straight to that rate because the information has been determined at least one RTT ago [172]. An endsystem cannot know how much a change in its own rate will affect the path, and the path might become congested in less than one RTT, too. Due to these effects it is impossible to design a perfect fast startup mechanism that performs well in all possible situations. Still, several network-supported congestion control mechanisms have been developed recently. Solutions range from TCP enhancements, such as Quick-Start TCP [RFC 4782], to new congestion control frameworks for a Future Internet. These protocols are presented in Section 4.5 and Section 4.6.

Most fast startup mechanisms can be characterized by four phases that are shown in Figure 4.17: During the (optional) *sensing* phase, some path characteristics are determined, potentially using signaling. Then the sender starts to send data (*probing*). The *validation* phase starts when corresponding ACKs arrive. The sender can then determine whether the initial choice was reasonable. Finally, the sender switches to the continuous congestion control (*continuation*), typically after the last ACK for the initially sent data has been received. The following sections analyze different realization alternatives of these phases.

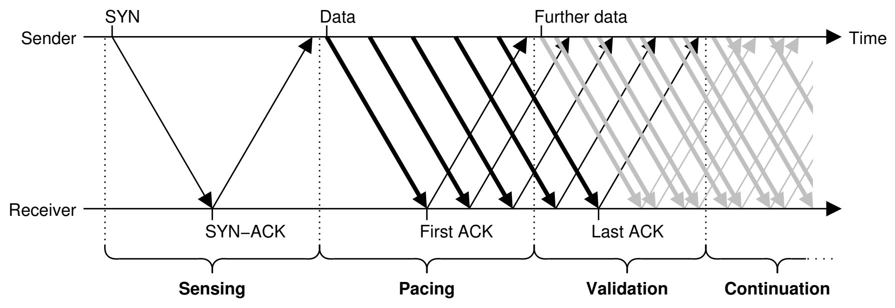


Figure 4.17: Main phases of a fast startup mechanism

4.4 End-to-end fast startup mechanisms

4.4.1 Existing Slow-Start enhancements without speedup

4.4.1.1 Improved transition to Congestion Avoidance

There are numerous proposals for Slow-Start enhancements that still start with a small initial window, but subsequently change the behavior. The most important approaches are classified in Figure 4.16; a more comprehensive list is given in Table 4.7.

The multiplicative increase of the Congestion Window during Slow-Start results in very large increments when CWND is already large. If a connection is able to use CWNDs of thousands of segments, the current Slow-Start procedure doubles this value in a single RTT. If the CWND exceeds the BDP, this can result in thousands of segments being dropped (*Slow-Start overshoot*), which is likely to trigger a retransmission timeout. In order to avoid this situation, “Limited Slow-Start” has been developed as an experimental TCP extension [RFC 3742]. It consists of a simple modification: If CWND exceeds a certain threshold, the sender limits the number of segments by which CWND is increased. The recommended threshold is 100 MSS. Limited Slow-Start thus subdivides the Slow-Start into an aggressive and a careful increase phase.

A drawback of this scheme is that it uses a fixed threshold. There have been various efforts to design a more intelligent transition from Slow-Start to Congestion Avoidance. They try to find the optimal transition point by bandwidth estimation techniques, taking advantage of the Slow-Start rounds that are quite similar to packet trains (see Section 4.1.5.1). Reference [92] proposes to use a packet pair bandwidth estimation for the first packets, and then to set the initial

Table 4.7: Comparison of selected end-to-end Slow-Start enhancements

	Enhanced Slow-Start			**Optimistic fast startup**						
	Limited SS [RFC 3742]	**Paced Start** [94]	**Hybrid SS** [81]	**Initial-Start**	**Rate-Based Pacing** [235]	**TCP Fast Start** [169]	**Swift-Start** [173]	**TFRC Faster Start** [120]	**Jump-Start** [136]	**Mega-Start**
Fast start	No	Poss.	No	**Yes**	Yes	Yes	Yes	No	**Yes**	**Yes**
Fast restart	No	Poss.	No	**Yes**	Yes	Yes	Yes	Yes	**Yes**	**Yes**
CA transit.	Yes	Yes	Yes	**No**	No	Poss.	Yes	No	**No**	**No**
Rate pacing	No	Yes	No	**No**	Yes	Yes	Yes	Yes	**Yes**	**Yes**
Bandw. est.	No	Yes	Yes	**No**	No	No	Yes	No	**No**	**No**

value of the SST to the estimated BDP. A correctly estimated SST value would eliminate the problem of numerous packet losses at the end of Slow-Start. Furthermore, Allman compared different estimators for setting an initial value of SST [5]. But he observed that they all perform poor due to effects caused by delayed acknowledgments and so-called “ACK compression”.

Another similar scheme is “Paced Start” [94]. During the Slow-Start, it uses packet train bandwidth estimation. Once a reasonable estimation is found, the CWND and SST are set to the estimated available bandwidth. This makes the transition to Congestion Avoidance faster than Reno and can avoid the overshooting problem. The recently developed “Hybrid Slow-Start” [81] also measures the arrival dates of ACKs and sets the SST to the estimated BDP. “Hybrid Slow-Start” requires only small changes in the sender-side TCP implementation and is integrated in new releases of the CUBIC congestion control in the Linux stack. There are also other similar proposals such as the one in reference [121]. Corresponding surveys can be found in [94, 81].

4.4.1.2 Information sharing between connections

An endsystem maintains state information for each TCP connection in a data structure that is called *TCP control block*. It contains information about the connection state, its associated local process, and congestion control parameters. This information can be maintained for each connection independently. But if there are several connections between the same endsystem, it could make sense to share information. *Information sharing* can be performed in two different ways: *Temporal sharing* reuses state variables of terminated connections, whereas *ensemble sharing* reuses state variables of existing connections. The combination of both mechanisms is known as *TCP Control Block Interdependence* [RFC 2140] and partly implemented in many TCP stacks. For example, as mentioned in Table 4.4, the Linux stack implements temporal sharing by a destination cache and stores the SST and RTT estimation variables. The cached values are used when another connection is established to the same destination IP address [189].

The *congestion manager* [RFC 3124] extends this concept. It provides a framework for a single, centralized entity in an endsystem that realizes congestion control on a per-host-pair rather than a per-connection basis. The idea is to integrate the congestion control management across all applications and all transport protocols. The congestion manager maintains the corresponding parameters and provides an API that enables applications to learn about network characteristics, to pass information to the congestion manager, to share congestion information with each other, and to schedule data transmissions. With the congestion manager, a new TCP connection could start with a high initial CWND if it shares the path with another TCP connection that is controlled by the congestion manager and has already reached a large CWND. More recent work has further extended this approach towards a common congestion controller that manages ensembles of TCP connections [194]. However, the architecture and APIs of a congestion manager entity have never been completely specified. As a result, congestion managers have never been widely implemented and used so far.

4.4.2 Existing optimistic fast startup mechanisms

4.4.2.1 Design alternatives

The Slow-Start starts with a small initial window and then increases the window exponentially, i. e., it starts carefully but is then very aggressive. This strategy can avoid multiple packet losses if a path is congested, but it also results in inefficient delays if the path has a large available bandwidth. In contrast, a disruptive *end-to-end fast startup* optimistically assumes that a path is not severely congested by default. Such an end-to-end flow start startup mechanism could use some of the following information that is often available in endsystems: The RTT, cached state variables for this destination, observable application communication characteristics (such as the amount of queued data in the socket), the local interface capacity, or application requirements.

4.4.2.2 Fast startup without burstiness control

The most trivial fast startup mechanism is just to increase the initial Congestion Window to a larger value than allowed by [RFC 3390], without any further modifications of the TCP congestion control. In this work, the approach of increasing the initial window is labeled *Initial-Start* (IS). Depending on the value of the initial window, this scheme may also have to modify the TCP flow control in order to be effective, as discussed in Section 5.2. There is some empirical evidence that some Web servers indeed use such an increased initial window [150].

Larger initial windows can result in disadvantages both for the individual connections as well as for the network [RFC 3390]. From the perspective of the endsystem, the risk of retransmission timeouts increases if multiple packets get lost. In the network, the packet drop rate can increase, in particular when drop-tail buffers are used, which have problems to absorb large packet bursts. This can reduce the efficiency: Packets that get dropped at some bottleneck result in wasted bandwidth on the path towards that congested network component. This is a problem if there are multiple congested links on the path, but this is a rather rare situation [RFC 3390]. Obviously, there can also be negative effects on other traffic that traverses the same bottleneck.

Another class of solutions starts with a small initial window, but then increases CWND much faster. This idea has been discussed various times. For instance, the Hybla congestion control flavor [39] suggest an increase algorithm that is independent of the RTT. For a baseline RTT of 25 ms, the behavior corresponds to TCP Reno. The larger the RTT, the faster CWND is increased during Slow-Start. This proposal specifically targets satellite environments and therefore does not address the interworking with other flow startup algorithms. It has not gained wider acceptance so far.

4.4.2.3 Fast startup with rate pacing

A larger initial window inherently increases the burstiness of the traffic. A *burst* can formally be defined as a sequence of consecutive packets with inter-packet gaps not greater than a specified parameter. Being a window-based protocol, TCP inherently sends packets in small bursts, which are also called *micro-bursts* [22]. Whether bursts cause problems or not depends on the burst-tolerance of the links on a path. The *maximum burst size* defines the maximum amount of bytes that a network component can absorb without dropping a packet. It largely depends on the buffer size and management strategy. According to [RFC 3390], bursts of up to 4 kB are not considered to be problematic. Empirical Internet measurements [22] show that larger bursts of the order of ten consecutive segments are not very frequent, and that the probability of losing one of these segments is low. This indicates that moderate burst sizes are unlikely to cause problems in the Internet.

Micro-bursts can be mitigated by *rate pacing*, which evenly spreads the transmission of a window of segments across a certain duration of time. Motivated by the fine timer resolutions in newer operating systems, the usage of pacing in TCP has been proposed several times since the late 1990ies. The permanent usage of rate pacing is a controversial topic. Several simulation studies claim that rate pacing could improve the throughput of TCP flows. But it has also been shown that TCP with permanent rate pacing may result in lower throughput and higher latencies, since rate pacing delays congestion signals [3].

Rate pacing could be useful to initialize the self-clocking mechanism during startup events, either for flow start or for flow restart. Rate pacing for TCP has first been proposed for the *fast restart* after idle times in order to improve the performance of HTTP version 1.1 sessions ("Rate-Based Pacing") [235]. A *fast startup* scheme with rate pacing is the "TCP Fast Start" proposal [169]. Instead of using Slow-Start it suggests to use temporal information sharing of the CWND and RTT estimation and to (re-) start downloads with the cached values. In order to avoid bursts, segments are clocked out by rate pacing. The authors also propose to assign the packets sent during the fast startup to a traffic class with higher drop probability.

Another proposed fast startup scheme is "Swift-Start" [173]. It combines packet-pair bandwidth estimation and rate pacing techniques. An initial CWND of four segments is used to estimate the available bandwidth along the path. After one RTT, this estimate is used to increase the CWND to a fraction of the measured BDP. As the window increase can be very large, rate pacing is used to avoid large bursts. However, it is unclear whether four segments are sufficient for an accurate estimation [5], and measurements have shown that the Swift-Start scheme is often outperformed by TCP Reno's Slow-Start.

Fast startup and fast restart mechanisms are also very relevant for multimedia streaming applications that have to quickly fill their playout buffer after startup. As the user must wait until the playout buffer is filled, a minimal initial buffering time is very important. In order to achieve this, the source must send data faster than the actual encoding rate, which contradicts the design philosophy of the Slow-Start. There is empirical evidence that widely used multimedia applications use a fast start and are TCP-unfriendly during the initial buffer period [164]. There are also ongoing efforts to develop a fast restart mechanism for flows using TFRC [120]. It allows TFRC flows to ramp up faster than TCP's Slow-Start after idle times.

4.4.3 Design of enhanced and new optimistic fast startup schemes

4.4.3.1 Jump-Start TCP

One possibility to cope with the potential aggressiveness of optimistic fast startup schemes is to take additional application characteristics into account. The simplest solution makes the startup dependent on the amount of application data that is available in the socket buffers. This idea has been proposed recently by a group around Allman [136]. The fundamental principle of this so-called *Jump-Start* (JS) is simple but disruptive: Instead of beginning with a small initial window, a sender just plays out the queued data during the first RTT using a rate pacing mechanism. Unlike other end-to-end fast startup mechanisms, the startup behavior of Jump-Start thus depends on the strategy how the application delivers data to the network stack. The more data is available, the higher is the initial rate. Jump-Start can be implemented as a sender-side modification only.

Jump-Start explicitly risks to start with a too large data rate, which could result in multiple lost packets. The original description [136] also proposes an orthogonal modified error recovery procedure after the validation phase (cf. Figure 4.17): If packet losses occur during the vali-

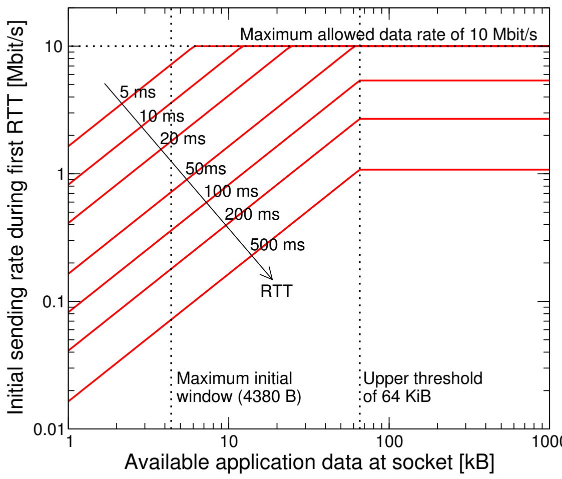


Figure 4.18: Modified Jump-Start with upper bounds for data rate and outstanding data

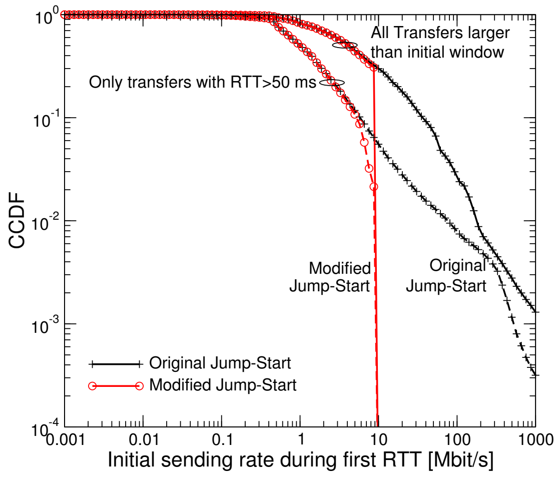


Figure 4.19: Initial rate of Jump-Start in the workload traces of Section 3.3.3.2

dation phase, the sender counts the TCP retransmissions. At the end of the loss recovery, the CWND is adjusted to roughly half of the number of successfully transmitted segments:

$$W_{\text{recov}} = \max\left(\frac{N_{\text{pacing}} - N_{\text{rtx}}}{2}, 1\right) \tag{4.11}$$

Note that the Equation (4.11) must be modified compared to the version in [136] in order to avoid negative values after many retransmissions. N_{pacing} is the number of segments sent during the rate pacing phase, and N_{rtx} is the number of retransmissions out of these segments. Reducing the CWND to roughly half of the load that the network was able to support corresponds to the reaction of a long-lived TCP Reno connection in a similar situation. Compared to an unmodified error recovery, Equation (4.11) may result in a significantly larger CWND, in particular after retransmission timeouts.

With Jump-Start, data transfers can use an initial sending rate that is significantly larger than the one of the Reno Slow-Start. Yet, Jump-Start has two properties that reduce the aggressiveness:

- *RTT differentiation*: For a given amount of data, the initial sending rate of Jump-Start depends on the RTT. The larger the RTT, the smaller the initial rate. This dependency, which can also be observed in Figure 4.18, has two advantages: The path capacity and RTT are often correlated. In typical LAN scenarios the capacity is large and delays are very small. In contrast, when a connection passes MAN or WAN links, the available bandwidth is likely to be smaller, and the delay is larger. In addition, the risk of passing through a congested link increases with the length of a path. Because of these effects, it could be reasonable to be less aggressive for larger RTTs. Furthermore, a congested link with filled buffers is likely to have a larger RTT, too. Thus, Jump-Start is inherently less aggressive when a link on the path is congested, which can be a desirable behavior. However, this also results in RTT unfairness, i. e., one of the shortcomings of the existing TCP flow startup mechanism still exists (cf. Section 4.3.2).
- *Data size dependency*: In the original Jump-Start idea there is a linear relationship between the data transfer size and the initial sending rate, which is also shown in the left part of Figure 4.18. Therefore, Jump-Start's aggressiveness is reduced by the typical distribution of transfer sizes in the Internet: As mentioned in Section 4.3.2.2, most of today's

TCP connections only transmit few data. In this case, Jump-Start does not start significantly faster than TCP Reno. This effect can be confirmed by analyzing the traces that are already introduced in Section 3.3.3.2. In combination with the measured RTT, one can estimate the initial sending rate that would be used by these larger transfers. According to the results shown in Figure 4.19, most transfers would use an initial rate of the order of few Mbit/s, which is large but often available in the current Internet.

The paper [136] also mentions further possibilities to cope with Jump-Start, including the usage of AQM to absorb busts or a lower-priority marking of packets sent during a Jump-Start.

4.4.3.2 Suggested modifications to Jump-Start

In the original proposal [136], N_{pacing} may be arbitrarily large if the RTT is small and if there is a large amount of data s in the socket buffer. According to Figure 4.19 there is a non-negligible probability of very high initial sending rates. As to be expected, this share is smaller among connections with a large RTT, but still significant. In order to prevent arbitrarily high initial rates and to reduce the aggressiveness, the author of this works suggests two modifications of the Jump-Start algorithms.

First, there should be an upper limit K_{data} for the amount of data sent in the first RTT. The suggested value for this threshold is $65,535\,\text{B}$. In addition to limiting the maximum burst size, this threshold also avoids interactions with receive window scaling (cf. Section 5.2).

Second, the maximum playout rate should be limited to a maximum rate K_{rate}. The suggested maximum value is 10 Mbit/s, i. e., the slowest rate of Ethernet links. If an endsystem knows that the capacity of an outgoing link is smaller than 10 Mbit/s, it would make sense to use this capacity as upper bound instead. In addition to avoiding extremely large initial rates, this upper bound also ensures that rate pacing is possible. Modern operating systems use a timer granularity of the order of milliseconds, which means that the TCP/IP stacks have difficulties to precisely schedule data transmission at rates much larger than 10 Mbit/s (cf. Section 5.3.2.2).

With the thresholds K_{data} and K_{rate}, the initial rate of the "Modified Jump-Start" scheme is

$$Q = \min\left(\frac{\min(s, K_{\text{data}})}{\tau}, K_{\text{rate}}\right). \tag{4.12}$$

This function is sketched in Figure 4.18 for different RTT values. With the selected parametrization, the initial maximum sending rate over a path with an RTT of 200 ms is about 2 Mbit/s, which is a realistic value in many scenarios. Figure 4.19 also shows that the two thresholds prevent extremely large initial data rates. Even the usage of K_{data} alone would already have a significant effect. Unless otherwise specified, in the following the term "Jump-Start" always refers to the scheme with the explained modifications.

4.4.3.3 A new, alternative fast startup scheme with explicit activation

Another promising alternative end-to-end fast startup congestion control scheme could provide a new interface to applications that allows them to activate selectively a fast startup. Otherwise, Reno's Slow-Start is used by default. This new idea combines principles of Jump-Start and Quick-Start TCP: Similar to Quick-Start, the applications can explicitly choose a reasonable initial rate Q_{req}. Unlike Quick-Start, this rate is just used without asking routers for approval:

$$Q = Q_{\text{req}} \tag{4.13}$$

The mechanism could work well if the requested rate is of the order of magnitude of the available bandwidth. Obviously, it will result in packet drops if the selected rate is too large. Similar to Jump-Start, a modified error recovery procedure can then be used. This new flow startup principle is labeled *Mega-Start* (MS) in this work.

The actual realization of the proposed Mega-Start scheme is rather straightforward, since it is basically a combination of two already known mechanisms, i. e., the application interfaces definitions are basically the same like in Quick-Start, while the internal algorithms are similar to Jump-Start. However, potential usage and impact of the Mega-Start proposal is different to other fast startup schemes: The fundamental idea of Mega-Start is that applications only activate the mechanism if a performance benefit is to be expected. As discussed in Section 5.1.1.2, there are several possibilities how an application could make an informed choice of the initial sending rate. If this mechanism was used only by a small share of applications, they could hardly cause harm in the Internet. One possible solution for such an activation strategy is proposed in Section 7.1, and its benefit is explained there, too.

4.4.4 Other approaches

A related mechanism has already been proposed earlier [254]. The idea is to use a performance gateway that monitors the traffic and stores information concerning larger IP subnetworks. Applications can communicate with this gateway by an out-of-band protocol and thereby obtain an estimate for the optimal initial CWND towards a given destination. Instead of Slow-Start, a pacing scheme is used to send out the packets. Simulations show that this scheme could significantly reduce the transfer times of short transfers [254].

A historical flow startup solution is *Transaction TCP* (T/TCP) [RFC 1644]. The experimental TCP extensions for transactions consists of a set of downward compatible mechanisms that intend to improve the performance of transaction-oriented client/server systems. T/TCP bypasses the three-way handshake at connection setup and avoids delays between connection release and the creation of a new instance of a connection. Even though the handling of the CWND in T/TCP is not precisely specified, information sharing could be used to enable fast startups. However, the T/TCP design has very fundamental problems. Omitting the three-way handshake makes T/TCP vulnerable to "SYN flooding" attacks and simplifies spoofing attacks. Also, an evaluation of the usage of T/TCP for HTTP-based application did not find significant performance advantage compared to persistent TCP connections [88]. T/TCP was supported by some TCP stacks in the mid-1990ies.

4.5 Network-assisted fast startup mechanisms

4.5.1 Overview of the Quick-Start protocol

4.5.1.1 Functional description of Quick-Start TCP

The *Quick-Start* (QS) protocol is a lightweight, coarse-grained, in-band, network-assisted fast startup mechanism. With Quick-Start, endsystems can rapidly determine an allowed sending rate in cooperation with the routers on the path, in particular at the beginning of a data transfer or after long idle times when the network conditions are unknown. Quick-Start is basically a performance enhancement for elastic best effort transport over paths with significant free capacity. It does only control the sending rate in such transient conditions. A fast startup is only allowed

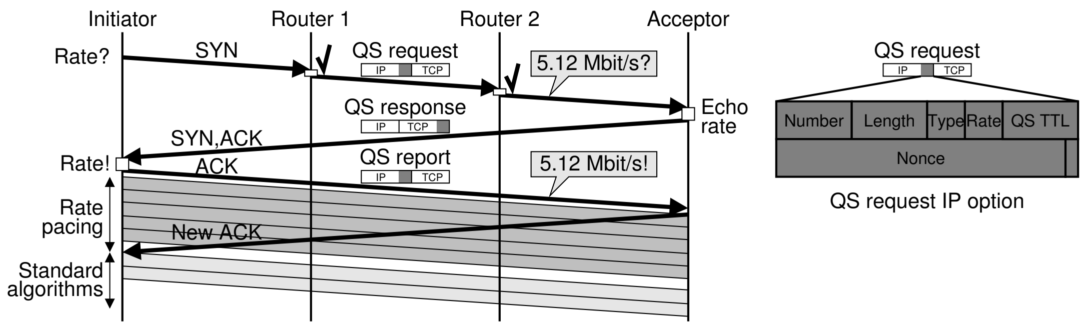


Figure 4.20: Quick-Start signaling during TCP connection setup

if there is a significant available capacity, i. e., the risk of causing congestion is rather small. Quick-Start can also be classified as an *anti-congestion control* scheme [192]. Quick-Start is originally specified as an experimental TCP extension [RFC 4782], which is complemented by a paper of Sarolahti *et al.* [192]. The IETF recommends that initial deployment of Quick-Start should be limited to controlled and trusted environments such as intranets. With some minor enhancements, Quick-Start can also be used in combination with SCTP and DCCP [63]. As the principal operation is the same in all cases, the following description only considers TCP.

The basic operation of the protocol can be explained with help of Figure 4.20, which illustrates a Quick-Start request during the TCP connection establishment. In order to indicate its desired sending rate, the connection initiator adds a "Quick-Start request" option to the IP header. This option is 8 B long and includes a coarse-grained target rate being encoded in a 4 bit field. A value of $i > 0$ corresponds to a rate $q = 40\,\text{kbit/s} \cdot 2^i$, i. e., there are 15 steps ranging from 80 kbit/s to 1.31 Gbit/s. The IP option also includes a QS TTL field and a QS nonce.

The routers along the path can approve, reduce, or disallow this rate request for q_{req}. Each router that supports the Quick-Start mechanism performs an *admission control* (or *approval control*, as explained later) and reduces or discards arriving request if there is not enough bandwidth available. If it processes the request, it also decrements the QS TTL field by the same amount it decrements the IP TTL field. If a router reduces the granted rate, it also randomly changes certain bits in the QS nonce. Routers that do not understand the request or that reject it can simply forward the packet without any processing.

If the request arrives at the destination, the granted rate is echoed back piggybacked as a TCP option ("Quick-Start response"), along with the received QS nonce and the difference between IP and QS TTL. The originator then determines whether all routers on the path support Quick-Start and whether all of them have approved the request. It therefore compares the echoed TTL difference with the difference in the original request. If they are not identical, there is at least one QS-unaware router on the path. In this case, the originator switches back to the default congestion control (i. e., Slow-Start) in order to ensure backward compatibility. By comparing the stored and the echoed QS nonce, the originator can also detect with a certain probability cheating attempts, i. e., receivers or routers that try to increase the rate above the granted value.

If the Quick-Start protocol grants a rate $q > 0$, the originator can increase its Congestion Window to the QS window $W_{\text{qs}} = \frac{q \cdot d}{MTU}$. The variable d is the measured RTT. W_{QS} may be much larger than the initial value allowed by [RFC 3390]. Provided that the responder announces a sufficiently large receive window that does not restrict the sender (cf. Section 5.2), the originator can start to send with the approved rate, using a rate pacing mechanism. The sender then

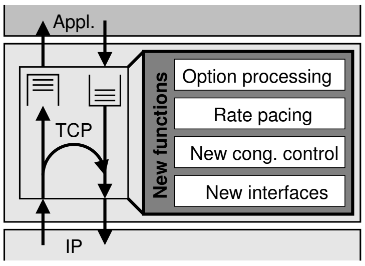


Figure 4.21: New TCP functions required by Quick-Start

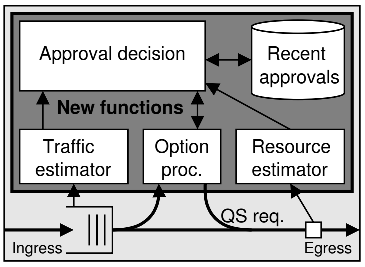


Figure 4.22: New IP functions required by Quick-Start

follows the principal phases of a fast startup shown in Figure 4.17. During a validation phase, it detects if all sent packets have successfully arrived at the receiver. If this is the case, the Quick-Start phase is completed after one RTT and the default TCP congestion control mechanisms are used for the subsequent data transfer. If packet loss is detected during the validation phase, the source must undo the CWND increase [RFC 4782].

4.5.1.2 Required functions in the IP and TCP layer

Quick-Start requires modification both in the transport layer of both involved endsystems as well as some changes in the IP processing in *every* router on the path. The new functions are depicted in Figure 4.21 and Figure 4.22, respectively. In the endsystem, the IP and TCP options must be processed, the rate pacing must be realized, and the congestion control and error recovery mechanisms must be adapted. Furthermore, new interfaces are required. The design of these interfaces is detailed in Section 5.1.1.

As shown in Figure 4.22, the support of Quick-Start also requires additional functions in the IP layer in every router on the path and also in the originator's IP stack: The IP stack must process the new IP options and perform an *approval control*, i. e., decide whether to accept a QS request and which data rate to grant. As discussed in the next section, this requires knowledge about the available bandwidth on the path, which can be obtained by determining the capacity of the outgoing links and monitoring their utilization. Depending on the control algorithm, other information such as the recently approved requests may have to be stored, too.

4.5.2 A new admission control concept: Approval control for Quick-Start

4.5.2.1 Constraints

The router functions required by Quick-Start have some similarity to MBAC, which is introduced in Section 4.1.5.2. Yet, there are unique constraints and differences between the *approval control* for network-assisted congestion control and traditional *admission control* algorithms:

- *Elastic traffic*: The approval control is performed for elastic traffic. Most MBAC algorithms assume inelastic traffic and are designed to meet a certain QoS objective, in particular a given buffer overflow probability. Buffer overflows are not a meaningful metric for elastic traffic. Even recent work on MBAC assumes that the number of flows is exactly known, and the developed algorithms are less efficient if this is not the case [78]. In IP networks it is impossible to precisely know the number of ongoing flows.

- *No guarantees*: Unlike IntServ-like QoS mechanisms, routers supporting Quick-Start are not required to make any reservations of bandwidth. They may *allocate* resources, but they are not required to *reserve* them for a flow, which would require per-flow state.
- *No traffic descriptors and traffic conditioning*: Many QoS admission control schemes assume a precise *a priori* statistical characterizations of the sources, e. g., by peak packet rates. The Quick-Start protocol does only request for an initial sending rate. Within few RTTs, the traffic pattern of the flow may be completely different and cannot be predicted for many TCP-based applications. In such an open environment, any admission decision can turn out to be wrong.
- *Flexibility of approval control*: Admission control typically realizes a decision with binary result. However, instead of denying a Quick-Start request, a router can also just grant a lower rate. This degree of freedom is not considered by most MBAC algorithms. Also, there is only a low penalty of not approving a QS request. The flows then just ramp up slower, but the network can still be utilized rather efficiently.
- *Unknown validity*: As Quick-Start is a sender-initiated scheme, a router cannot know whether a request will be approved by other routers down the path, or whether it will be denied. It even does not know whether an arriving request has already passed a QS-unaware router and is already implicitly denied. This issue could only be solved by a receiver-initiated scheme as shown in Figure 2.8, which would result in larger delays.

In summary, the mathematical assumptions of the statistical methods used by many MBAC schemes are not fulfilled. As a consequence, the proposed algorithms can hardly be applied in the context of Quick-Start. Instead, the Quick-Start approval control requires *uncomplicated algorithms*. Approval control mechanisms for congestion-controlled traffic is a rather unexplored field. There is related work on admission control for elastic traffic in order to enforce an upper limit on the number of TCP connections [73], but this is again a pure admission decision problem. In the context of Quick-Start, approval control mechanisms are only studied in the references [RFC 4782], [192], and [191]. Three different problems have to be solved: (1) estimation of the available bandwidth, (2) approval decision, and (3) handling of recent requests. The following subsections review the design space of these building blocks and then propose new algorithms.

4.5.2.2 Design space for estimation of the available bandwidth

Approval control requires an *online estimation* of the available bandwidth on the outgoing links. The Quick-Start approval control is performed by network components processing IP packets, which are not necessarily aware of the dimensioning of outgoing links and potential bottlenecks in the link layer technology. As stated in Section 4.1.5.1, an accurate and timely measurement of the available bandwidth is difficult, and probing techniques have disadvantages [RFC 5559]. Yet, the coarse granularity of the Quick-Start mechanism mitigates this problem: The Quick-Start approval control only requires a *rough approximation* of the available bandwidth (see Section 6.5.3). In case that the link capacity is rather constant, such an estimation can be obtained by determining the *link capacity* and subtracting the *link usage*, i. e., the carried traffic. The capacity of router interfaces can be determined for instance from the corresponding network technology. If the capacity is not constant (e. g., on shared wireless links), cross-layer information exchange may be required.

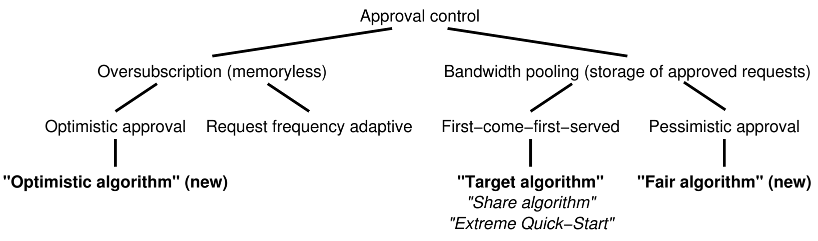


Figure 4.23: Classification of different approval control algorithms. The highlighted algorithms are evaluated in this work.

The measurement of the link usage requires an online rate measurement within a time-scale of the order of magnitude of the RTT of the flows. Such a frequent estimation of the carried traffic can be derived from measurements of the transferred data volume, either within disjoint time intervals or with a window-based solution [145]. The measurement can be combined with low-pass filters, such as the *Exponentially Weighted Moving Average* (EWMA), trend-based approaches [38], or forecasts. Sarolahti *et al.* proposed to use either an EWMA estimator that updates the estimate $u(t)$ of the link usage based on the measured value $x(t)$ by

$$u(t) \leftarrow \alpha_{\text{ewma}} \cdot x(t) + (1 - \alpha_{\text{ewma}}) \cdot u(t - \Delta), \tag{4.14}$$

or a *peak utilization* estimator

$$u(t) \leftarrow \max\left(x(t), x(t-\Delta), \ldots, x(t - N_{\text{peak}} \cdot \Delta)\right). \tag{4.15}$$

The peak estimator records the carried traffic during the most recent N_{peak} disjoint time intervals of duration Δ; the suggested value is $N_{\text{peak}} = 5$. The peak estimator determines a more conservative estimation of the available bandwidth, as it reacts fast to sudden increases, but also remembers periods of high utilization in the recent past. It is the default estimator in this work as explained in Section 6.3.3.2.

4.5.2.3 Design space for approval decisions

There are fundamentally different possibilities how an approval algorithm can be designed. As classified in Figure 4.23, one can either use *oversubscription* or *bandwidth pooling*[3]. In the former case, each request is approved independently from other requests. Possible algorithms could for instance always approve a rate up to the currently available bandwidth. Such an *optimistic algorithm* may grant a large amount of bandwidth if many requests arrive, i. e., it is vulnerable to flash crowd effects. But, unlike end-to-end fast startup schemes, such an approval control still reduces the risk of congestion, since it can prevent further aggressive flow startups if a link is already congested. Alternatively, router algorithms could apply a more complex control law that takes additional statistics into account, such as the queue length, similar to the mechanism introduced in Section 4.6.2. A simple *flash crowd protection* could also be realized by measuring the average number of requests within a certain time duration and reduce the maximum granted bandwidth if it exceeds a threshold.

An alternative solution is to manage the available bandwidth as a pool of resources and allocate a certain part of the pool resources whenever a request is granted. This avoids oversubscription, but it requires a history of recently approved requests. Examples for such algorithms are

[3]The term *resource pooling* is also used in multipath transport in a completely different context.

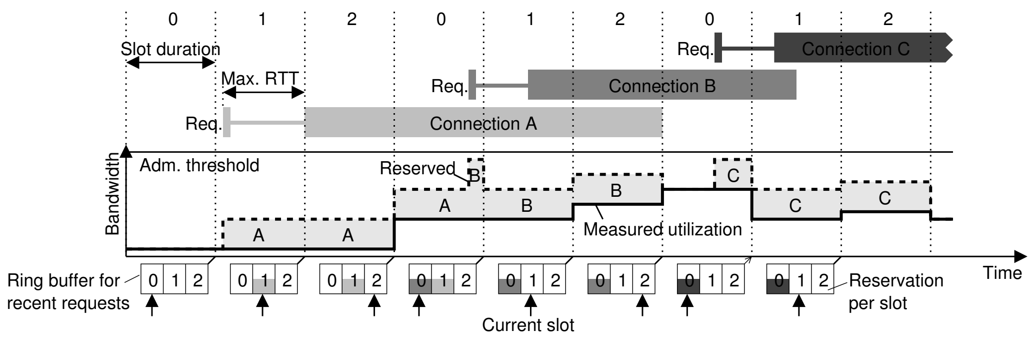


Figure 4.24: Illustration of the Quick-Start approval control and recent request storage

presented in Section 4.5.2.4 and Section 4.5.3.3. The problem of considering already admitted flows also occurs in classic admission control algorithms [29].

There is a trade-off between oversubscription and bandwidth pooling: A conservative approval control might deny too many requests, resulting in larger flow completion times, while an optimistic scheme risks temporary congestion if many new flows arrive in parallel. There are also further degrees of freedom how to deal with requests that exceed the bandwidth that can be granted. For instance, the granted rate could depend on the request rate. A further option is just to deny requests that exceed the available bandwidth, instead of reducing the rate. This strategy would give incentives to hosts not to request for unnecessarily high data rates [199].

Keeping track of recently approved requests could be realized by storing temporal per-flow state, e. g., between the “Quick-Start request” and the “Quick-Start report” [192]. But this solution would require a potentially very large table of per-flow soft state. A more scalable solution is to keep track of the *aggregate approved rate* over recent link usage measurement intervals. This does not require per-flow state. In [199], the author of this work proposes to use a ring buffer of size Ω for the storage of the recently granted aggregate bandwidth. Ring buffers are well suited for storing reservations [37]. The usage of a ring buffer with $\Omega = 3$ is illustrated in the bottom part of Figure 4.24: Whenever a request is granted, the granted rate is added to the current field of the ring buffer. After the time interval Δ, the pointer to the current element is shifted to the next element and its value is zeroed.[4] The aggregated allocated bandwidth results from the sum of all elements in the ring buffer.

Figure 4.24 also illustrates that the rate measurements and the history of recent approvals must be coordinated: When a request is approved, the granted capacity must be stored until a new rate sample $x(t)$ is measured and until it is almost certain that this sample includes the data rate of the new flow. This requires rather frequent rate measurements, i. e., Δ must be of the order of the RTT of the flows. In the example of connection B in Figure 4.24, it takes three rate measurements after the request until the router can forget the Quick-Start request. Choosing the configuration parameter Ω results in a trade-off: If it is too small, the router forgets recent rate approvals too quickly and thus may oversubscribe the available bandwidth. But if Ω is too large, the router is too conservative when approving Quick-Start requests. The required number of required ring buffer spaces is derived in [199] as

$$\Omega_{\text{opt}} = \left\lceil \frac{d_{\max}}{\Delta} + 2 \right\rceil \tag{4.16}$$

[4]This assumes that both rate measurements and the ring buffer are updated after time intervals of duration Δ. This is the most simple realization alternative because then all information is recalculated at the same time. Theoretically, both procedures could be decoupled, but this alternative is not further considered in this work.

if a new rate measurement should incorporate a flow starting with the granted rate, and if the worst-case RTT is $d_{\max}$. An optimistic setting is $\Omega = 2$, which is also used in [192, 191],

4.5.2.4 *Existing algorithms*

The Quick-Start specification [RFC 4782] does not explicitly enforce a specific approval algorithm. Sarolahti [191] introduces and compares three different solutions:

- The "target algorithm" uses bandwidth pooling. It approves requests up to a configured percentage of the link's bandwidth minus the sum of the current link usage and the aggregated bandwidth of recently granted requests (*unused bandwidth*). This algorithm is similar to the well-known "measured sum" admission control strategy [29].
- The "share algorithm" allocates a pre-set fraction of the unused bandwidth for each arriving request. The unused bandwidth is determined by the link capacity minus the used traffic minus the recent QS approvals similar like in the previous case.
- The "extreme Quick-Start" solution maintains per-flow state about QS requests. The router keeps track of each individual QS request and report. Therefore, it can more precisely estimate the bandwidth that has been approved but that is not used yet. The proposed algorithm calculates for each flow how much data has been transmitted after a QS request. This solves the problem of wasted capacity if the history of recent approvals stores a request for a too long time. Furthermore, a score is used to identify senders that tend to request more bandwidth than they actually use.

The large amount of state information kept by the "extreme Quick-Start" algorithm impose severe implementation challenges, and it is therefore unrealistic that all routers implement it. It could, however, be used at edge nodes, since it offers some protection against single hosts issuing many large QS requests without using them [191].

[RFC 4782] argues that the target algorithm would be a reasonable solution. Its main advantage compared to the "share" algorithm is that it includes a threshold θ up to which a link is considered "underutilized". With this threshold, the *unused bandwidth* available for Quick-Start is $a_{\mathrm{QS}} = \theta \cdot c - u(t)$, if c is the assumed link capacity. It grants requests up to the maximum rate

$$q_{\mathrm{target}} = \min\left(q, a_{\mathrm{QS}} - H(\Omega)\right), \tag{4.17}$$

where $H(\Omega)$ is the sum of all stored previous requests. A flowchart of the algorithm and a possible integration in the IP option processing is presented in Figure 4.25. Due to the rough granularity of the rate field, the actually granted rate q_{approved} must be one of the 16 values. If one selects the largest possible value $q_{\mathrm{approved}} \leq q_{\mathrm{target}}$, the granted rate is reduced in average by factor 1.5.

4.5.3 Design of improved approval control algorithms for Quick-Start

4.5.3.1 *Motivation*

The target algorithm has two shortcomings: First, granting resources only on a FCFS basis is inherently unfair. A single request for a large rate (e. g., 1.31 Gbit/s) could allocate all bandwidth and cause all further requests to be denied during a time interval of $\Omega \cdot \Delta$. This unfairness of the target algorithm is empirically observed in [191], but not solved. The unfairness is even worsened by a specific rule of [RFC 4782]: If a Quick-Start requests gets denied, the sender

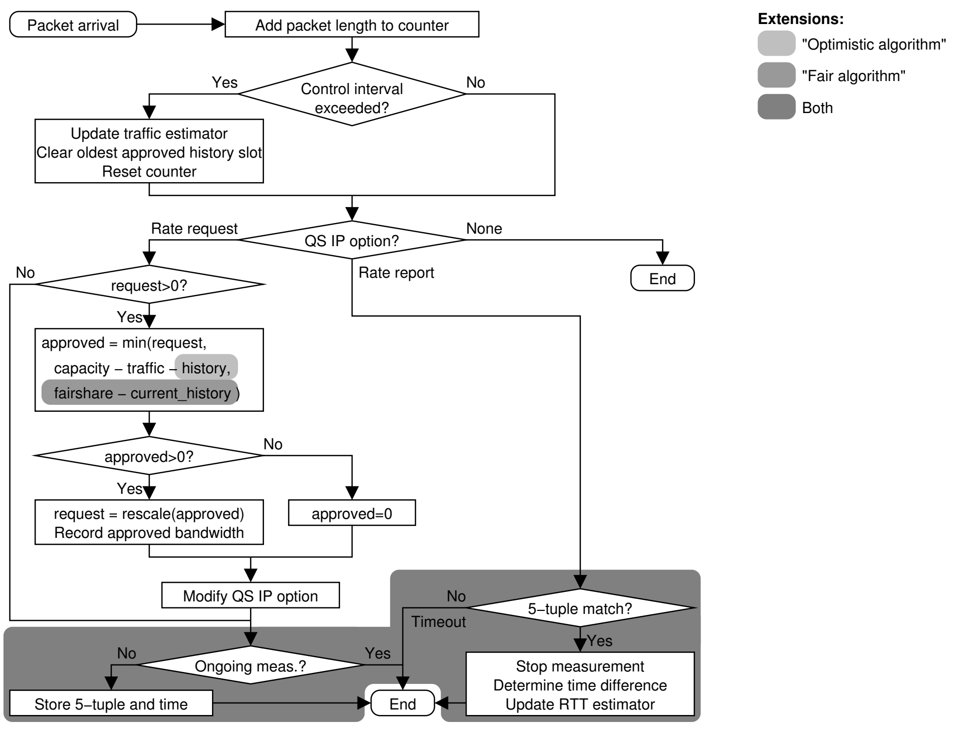


Figure 4.25: Quick-Start approval control by the “target algorithm”. The proposed extensions by the “optimistic algorithm” and the “fair algorithm” are highlighted by shaded boxes.

must revert to the CWND that was valid before the Quick-Start request. This means that the CWND after a denied Quick-Start request may be smaller than the CWND that would result from the Slow-Start algorithm.

Second, the performance depends significantly on the estimation interval Δ. Sarolahti *et al.* [192, 191] use a fixed value of $\Delta = 150\,\text{ms}$ in their studies, but never address how this parameter should be set. In the following, two enhanced Quick-Start router algorithm are presented that address these two issues. They have first been presented by the author in [206].

4.5.3.2 Optimistic algorithm

One possible solution to address the FCFS unfairness problem is not to use the bandwidth pooling philosophy. Specifically, a router could approve each request independently of previously approved requests, just considering the available bandwidth. The granted rate of this “optimistic algorithm” is

$$q_{\text{optimistic}} = \min\left(q, a_{\text{QS}}\right). \tag{4.18}$$

Obviously, this algorithm can oversubscribe link capacity, and it thus risks congestion if many requests arrive within a short time interval. Yet, the optimistic algorithm protects a link against an *aggravation of existing congestion situations*, since requests do not get approved if a link is already completely utilized. Its main advantage is the simplicity: Requests can be approved without modification of global system variables, which avoids the synchronization issues discussed in Section 5.3.3. In fact, the design philosophy of the “optimistic algorithm” is similar to the one of other network-controlled schemes introduced in Section 4.6.3.

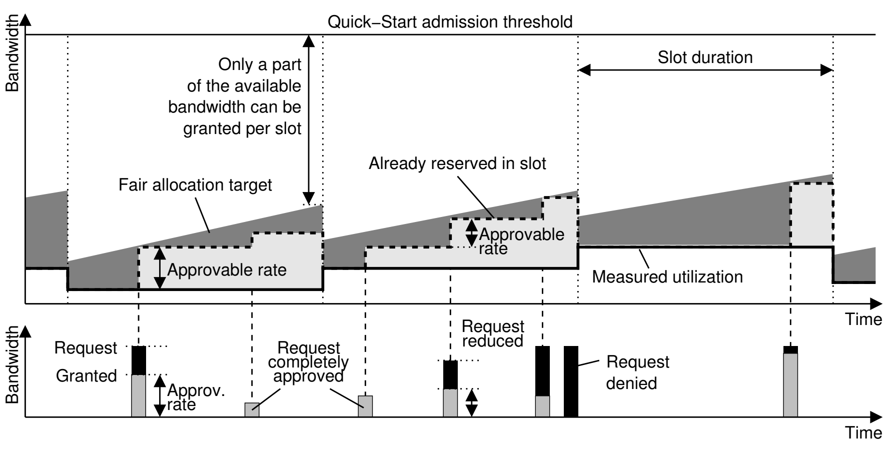


Figure 4.26: Illustration of the approval control by the "fair algorithm". Within one slot, the fair allocation target increases linearly with the time. Requests may preventively be reduced or denied.

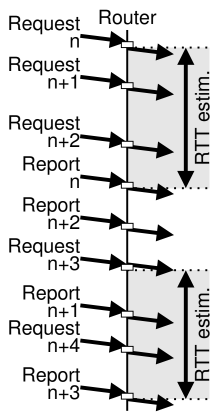


Figure 4.27: RTT estimation method for Quick-Start

4.5.3.3 *Fair algorithm*

If bandwidth pooling is desired, an ideal max-min fairness of the granted rates could in theory be achieved as follows: If the number of Quick-Start requests within a time period $\Omega \cdot \Delta$ was known, one could grant to each of them its corresponding share of a_{QS}. However, correctly predicting the number of future requests is impossible in practice unless the arrival pattern is very regular. It would also require additional state variables compared to the target algorithm.

The proposed "fair algorithm" does not try to predict the number of requests. Instead, it uses the elapsed time t since the last rate measurement t_0 as an additional parameter. The fundamental idea is to reserve a certain share of resources for requests that may arrive in future. As shown in Figure 4.26, the algorithm maintains an additional fair allocation target rate

$$q_{\mathrm{fair}} = \min\left(\frac{1}{\Omega}\left(\Theta \cdot a_{\mathrm{QS}} + (1-\Theta) \cdot a_{\mathrm{QS}} \cdot \frac{t-t_0}{\Delta}\right) - H(0), q_{\mathrm{target}}\right) \tag{4.19}$$

within each time slot of duration Δ if there are previous requests. The sum of the already approved rates in this slot is $H(0)$. The maximum amount that can be granted within one slot is upper limited by a_{QS}/Ω. This increases the likelihood that requests arriving during the next slots will find non-allocated capacity. Furthermore, within one slot, the fair allocation target is linearly increased to prevent a single request randomly arriving at the begin of the interval to allocate all resources. The parameter Θ, which is set by default to 0.125, still allows such requests to obtain some reasonable amount of bandwidth. Equation (4.19) can easily be integrated in the target algorithm, for which very efficient implementations exist (cf. Section 5.3.3).

4.5.3.4 *Proposal of an adaptive parametrization*

The other open issue is how to appropriately set the measurement interval Δ. Setting this parameter has to cope with a fundamental trade-off: On the one hand, if Δ is too short, the measurement values may include a significant jitter. On the other hand, if it is too long, the measured rate might already be outdated once it is calculated. As derived in reference [101], if the rate is modeled by a random process, the variance of the sample decreases with the length of the averaging interval Δ. Furthermore, if Ω is set to a constant value, the parameter Δ also determines how long approved requests are stored. According to Equation (4.16), Δ should be of the order of magnitude of the RTT of the flows. If Δ was significantly larger, the approval control would remember requests too long and may unnecessarily deny subsequent requests.

There is no single optimal value for Δ. Research results on network-controlled admission control show that their control interval should be set between the average and maximum RTT of the flows through the router. A further proposed extension of the Quick-Start approval control uses this idea and *automatically tunes the measurement interval according to the RTTs* of the QS-enabled flows. This requires an estimation of the RTT of flows. There are various generic techniques how to passively estimate the RTT of TCP connections [106]. The most simple one keeps track of TCP's three-way handshake and estimates the RTT between the last <SYN> and the first <ACK> segment. However, a drawback of this approach is that it determines the RTT of all flows, even if they do not support Quick-Start. The solution proposed by the author explicitly considers connections with Quick-Start only: As depicted in Figure 4.27, an approximation of the average RTT can easily be obtained by measuring the delay between the Quick-Start request and the Quick-Start report IP options. If there is only one ongoing measurement, the router must only store a timestamp and a set of values that uniquely identify a flow. Figure 4.25 also shows that the integration of this method into the processing of the Quick-Start IP options is straightforward.

These different approval control algorithms are compared in Chapter 6. There are also other possibilities how to further enhance the approval control without giving up the simple controller design. For instance, one could try to use an overbooking factor based on past experience [154]. Such possible enhancements are left for further study.

4.5.3.5 Implications of the activation strategy of Quick-Start

The Quick-Start mechanism must be used carefully, as it can easily be rendered useless if applications request for a large data rate without using it. The problem of requested but unused bandwidth can to some extent be solved by the optimistic approval control in routers, but this solution risks bandwith oversubscription. Actually, Quick-Start requests should only be sent by endsystems if they make sense.

Useless requests can be avoided by an *intelligent activation* strategy: As Quick-Start only makes sense for mid-sized data transfers, it should only be activated if such a transfer is expected. This work argues in favor of an explicit fast startup activation interface between the network stack and the applications which is introduced in Section 5.1.1. This interface allows an application to trigger a fast startup only if it indeed anticipates a larger data transfer. This information could for instance be determined from the size of the data objects, or from payload identifies in application protocols. Alternatively, the network stack could internally monitor the socket buffer and use the queue length as decision criterion for sending a Quick-Start request.

In both cases, a simple activation heuristic would be to activate Quick-Start only if the *amount of data is larger than a certain threshold* χ. A reasonable setting of χ is determined in Section 6.4.3.3. This solution requires that the network stack could monitor the data arrival in the socket, similar to the Jump-Start proposal, or that explicit application knowledge is used. The latter variant is further studied in Section 7.1. If these heuristics fail and if there are many spurious requests, the utility of the Quick-Start protocol is reduced. This problem shows that the usage of the Quick-Start protocol is more complex than end-to-end fast startup congestion control schemes – not only because it requires modifications in routers.

Table 4.8: Comparison of selected fast startup TCP enhancements

	Initial-Start	**Jump-Start**	**Mega-Start**	**Quick-Start**
Type	End-to-end	End-to-end	End-to-end	Network-assisted
Sensing	Measure RTT	Measure RTT	Measure RTT	Explicit feedback
Probing	Large window	Play out appl. data	Use given rate	Use approved rate
Validation	Unmodified	Count retransm.	Count retransm.	Observe loss
Continuation	Unmodified	Adapt after loss	Adapt after loss	Revert after loss
Rate pacing	No	Yes	Yes	Yes
Activation by application possible	No	No	Yes	Yes

4.5.4 Other approaches

The Quick-Start mechanism is at the time of writing the most elaborated network-assisted congestion control scheme. Its differences to end-to-end fast start startup mechanisms are summarized in Table 4.8. A related approach is “AntiECN” [122], which suggests an additional one bit per-packet feedback in the IP header. By setting this bit, routers could inform endsystems that they consider a link to be underutilized and that a faster increase of the CWND is possible. The “Variable-structure Congestion Protocol” [250] suggests a similar notification by using the two ECN bits. Routers classify the level of congestion into three regions (low-load, high-load, and overload), and encode this state into the ECN bits. Based on the load region the sender uses different window increase or decrease algorithms.

The idea of granting a certain rate is also part of a proposal for a QoS control plane for military networks [18, 187]. It uses a new IP option with a length of 16 B that includes an available and a guaranteed rate, which are negotiated along the path with a sender-initiated request-response signaling. There is not much information about the current status of this protocol.

In [160], a network-assisted configuration of the parameters of the Limited Slow-Start is explored. And there are also several proposals for additional IP control plane protocols that collect bandwidth information from the routers along the path. One example is the “Performance Transparency Protocol” [246].

[RFC 4782] also extensively discusses design alternatives for the Quick-Start mechanism, in particular the usage of other signaling protocols such as ICMP or a QoS reservation protocol like RSVP or NSIS. Unlike IP options, this would result in an out-of-band signaling that has several disadvantages: The authentication of the messages would be much more complex, since it must be ensured that the information originates from routers on the path. Also, the traversal of NATs and IP tunnels is much simpler for in-band signaling with an “end-to-end echoing” feedback model. Many functions of complex protocols such as RSVP or NSIS are not required for network-assisted congestion control.

4.6 Network-controlled congestion control

4.6.1 Network control as a clean slate approach

Several new network-controlled congestion control schemes have been developed as part of the research activities on future *clean slate* Internet architectures. They explicitly do not consider

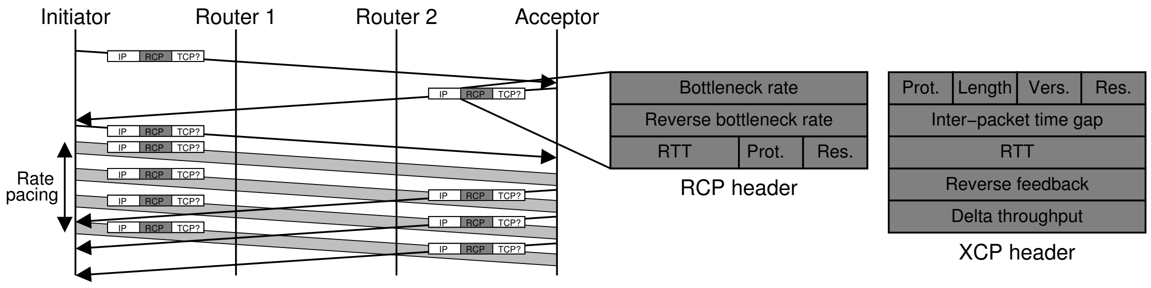


Figure 4.28: Operation of XCP and RCP with a new congestion header

backward compatibility to TCP/IP as a major design objective (cf. Section 2.4.3). An important outcome of this research is the idea of *putting additional information in every single packet*. On the one hand, this additional information informs network components about performance metrics of the flow (indication information). On the other hand, an on-the-fly modification of this information makes it possible that network components control a flow without requiring per-flow state. The insight that per-flow state can be avoided by transporting additional information in every packet dates back to work of Stoica *et al.* [221], who showed that IntServ-like rate guarantees are possible without per-flow management if precise deadline timing information are transported in every packet. This approach of transporting state in packets is called “Dynamic Packet State”. In more recent work [222], Stoica *et al.* develop an architecture for fair bandwidth allocation that uses a similar mechanism and transports information about the bandwidth in packet labels.

The idea of realizing network-controlled congestion control with help of an extended packet header has recently attracted considerable attention in the research community, as it is a promising approach to address the shortcomings of an end-to-end congestion control. Several new schemes have been proposed, which are still subject to ongoing research. The following two sections briefly review the two most promising protocols.

4.6.2 Overview of the eXplicit Control Protocol (XCP)

The *eXplicit Control Protocol* (XCP) is an experimental network-controlled congestion control protocol that has been developed by Katabi *et al.* [110]. It uses fine-grained, in-band, per-packet feedback from network components in order to improve the performance in networks with a high bandwidth-delay product. The XCP controller in network components decouples the efficiency control from fairness control. The design of XCP is expected to achieve flow-fair bandwidth allocation, a high link utilization even in large BDP environments with small persistent queue size, and near-zero packet drops. Thus, the fast startup is not a primary design goal. XCP requires some arithmetic operations in each queue on the path, but it does not require any per-flow state in the network. XCP is the outcome of a clean-slate research project [51] and still work-in-progress. An incomplete specification of the protocol is available [64].

XCP carries per-flow congestion state in every packets by using a *congestion header* with a length of 20B, which is added between IP and transport layer. Its structure is depicted in Figure 4.28. XCP only realizes congestion control, but no other transport protocol functions. It can thus be understood as a new *shim layer* that realizes congestion control independently of the transport protocol. The existing implementations use a modified TCP as transport protocol on top of XCP. In principle, any other window-based protocol could be used as well. In each packet, the XCP sender indicates how much it would like to increase or decrease its throughput ("delta throughput"), and this field is updated by the network components along the path. XCP uses *end-to-end echoing*: When a data packet reaches the receiver, this value is returned to the sender in the "reverse feedback" field of a congestion header of a returning packet.

Katabi *et al.* [110] precisely specifies the algorithms that each XCP-enabled network component has to execute on arrival and departure of packets. The network components monitor the input traffic rates to their output queues and their queue length and maintain some statistics. For each output queue, an XCP-enabled network element uses two algorithms for efficiency and fairness control: The efficiency controller is responsible for maximizing the link utilization and draining any standing queues. The fairness controller is responsible for fairly allocating the bandwidth to the flows sharing the link. These two algorithms are executed periodically after a control interval. The control interval duration Δ is set to the average RTT d_{avg} of the flows. The efficiency controller periodically calculates the desired change of the aggregated bandwidth:

$$a_{\text{XCP}} = \alpha_{\text{XCP}} \cdot (c - x(t)) - \beta_{\text{XCP}} \cdot \frac{b_{\text{persist}}(t)}{\Delta}. \tag{4.20}$$

In this equation c stands for the assumed link capacity, $x(t)$ is the bandwidth of the arriving traffic in the last control interval, and b_{persist} is the minimum queue length observed during the control interval. The first term calculates the unused bandwidth, whereas the second term ensures that a persistent queue is drained. The aggregate feedback a_{XCP} may be positive or negative. The fairness controller uses an AIMD principle to allocate the positive or negative feedback pools to flows: If $a_{\text{XCP}} > 0$, it equally assigns the spare bandwidth to all flows. If $a_{\text{XCP}} < 0$, the negative feedback is distributed proportionally to each flow's current throughput. As a consequence, if a link is fully utilized, XCP will not assign much capacity to a new flow. In order to overcome this problem, XCP uses a "bandwidth shuffling" mechanism that redistributes a small amount of the available capacity (at most 10 %) by adding it to the positive and negative feedback pools. The stability of the controller depends on the parameters α_{XCP} and β_{XCP}. A linear stability analysis [110] finds that a system with a single link is stable independently of delay, capacity, and the number of flows, assuming that the link capacity is known. The recommended parametrization is $\alpha_{\text{XCP}} = 0.4$ and $\beta_{\text{XCP}} = 0.226$ [110, 64].

XCP requires that each queue controller knows the exact capacity of its link. In shared access media, knowing the actual capacity of the channel is a difficult task. As analyzed in Section 6.5.3, any link capacity overestimation results in a persistent queue length, whereas link capacity underestimation causes a link underutilization. There have been efforts to address this shortcoming by modifications of the XCP control algorithms [1], which use the variation of the queue length instead of an estimation of the available bandwidth. However, these modifications are not very robust and reduce the convergence time to full utilization, and they can only operate if there is a certain persistent minimum queueing delay.

4.6.3 Overview of the Rate Control Protocol (RCP)

XCP has two shortcomings: First, the startup of flows is known to be rather slow, resulting in poor flow completion times for short flows [56]. Second, it requires per-packet calculation and manipulation of state variables, which results in computational overhead and synchronization problems among parallel packet processing entities [84]. The *Rate Control Protocol* (RCP) has been developed by Dukkipati [57] in order to provide a simpler alternative to XCP. It uses a per-packet feedback similar to XCP. Different to XCP, the design objective of RCP is to emulate process sharing and to minimize the average flow completion time. Furthermore, RCP only requires very simple per-packet computations.

An RCP-enabled network component maintains a single rate that is assigned to all flows that pass through it. RCP thus achieves automatically max-min fairness. The rate is updated once per control interval, which is set to the average RTT of the flows similar to XCP. In an experimental implementation [58], RCP is implemented as an own protocol layer between IP and transport layer with a congestion header having a length of 12 B. As shown in Figure 4.28, this header mainly consists of a rate field and a second field used by the receiver to echo the received value back to the sender. RCP assumes that senders use a rate-based sending mechanism.

By assigning a single rate to all flows, RCP optimistically assumes that not too many flows will arrive in the next control interval, and it tolerates larger instantaneous queue sizes. Processor sharing could easily be emulated if the number of ongoing flows $n(t)$ was known. The flow rate fair assignment of the assumed link capacity c would then be $a_{\mathrm{RCP}}(t) = c/n(t)$. However, since it is difficult to precisely estimate $n(t)$, RCP uses an approximation $n(t) \approx c/a_{\mathrm{RCP}}(t-\Delta)$. This empirically motivated, recursive estimation method assumes that in the last time interval the rate assignment has been perfect. Simulation studies [57] show that the estimator performs reasonably well if there are many long-lived flows, and that it is not critical that short-lived flows cannot be counted accurately. With this recursive approximation, RCP calculates the rate $a_{\mathrm{RCP}}(t)$ once per control interval as

$$a_{\mathrm{RCP}}(t) = a_{\mathrm{RCP}}(t-\Delta)\left(1 + \frac{\frac{\Delta}{d_{\mathrm{avg}}}\left(\alpha_{\mathrm{RCP}}\left(\theta \cdot c - x(t)\right) - \beta_{\mathrm{RCP}}\frac{b(t)}{d_{\mathrm{avg}}}\right)}{\theta \cdot c}\right), \tag{4.21}$$

where d_{avg} is the moving average of the RTT measured among the traffic passing through the RCP queue, Δ is the update interval duration with $\Delta \leq d_{\mathrm{avg}}$, $a_{\mathrm{RCP}}(t-\Delta)$ is the last used rate, c the assumed link capacity, $x(t)$ the measured input traffic rate during the last update interval, $b(t)$ the instantaneous queue size, and θ a parameter that selects a desired peak utilization ($0 < \theta \leq 1$). The configuration parameters α_{RCP} and β_{RCP} affect stability and performance: α_{RCP} represents a trade-off between stability and response time. A larger value results in faster response times at the expense of reduced stability margins and *vice versa*. β_{RCP} corresponds to the trade-off between acceptable queueing delay and the fair-share rate during transient periods. It can be shown that RCP is locally stable if certain conditions for α_{RCP} and β_{RCP} are fulfilled [57]. The recommended settings is $\alpha_{\mathrm{RCP}} \in (0.4, 0.6)$ and $\beta_{\mathrm{RCP}} \in (0.2, 0.6)$ [57], even though Dukkipati also uses $\alpha_{\mathrm{RCP}} = 0.1$ and $\beta_{\mathrm{RCP}} = 1.0$ in many studies. There is also ongoing work to develop improvements of RCP, such as replacing the queue term by a different controller [102].

4.6.4 Other approaches

Several other, similar network-controlled congestion control schemes have been proposed:

- "MaxNet" [225] is a congestion control approach where the source receives the maximum price on the path, i. e., the congestion level of the *most* congested bottleneck link. The source then chooses its sending rate to maximize its own utility function.
- In "JetMax" [256], routers calculate the target rate by estimating the number of flows that are bottlenecked at a link and by estimating the capacity used by non-bottlenecked flows. In steady state, this achieves max-min fair rate allocation in multi-link scenarios.

However, recent studies report that this approach cannot utilize links in presence of short flows, because the heuristics fail to differentiate between short and long flows [102].

- "Congestion avoidance with Distributed Proportional Control" is a network-controlled congestion control scheme developed by Welzl [245]. It uses the out-of-band "Performance Transparency Protocol" to retrieve feedback about the available bandwidth. The control laws are rate-based and independent of the RTT. They are an extended version of the *Available Bit Rate* (ABR) resource management in *Asynchronous Transfer Mode* (ATM). The scheme does not work well for short flows.
- The Available Bit Rate traffic control of ATM [ATM TM] periodically sends resource management cells from the source to the destination. They can support different feedback signaling schemes, such as a one-bit explicit congestion notification or an explicit rate feedback. The ATM switches calculate the maximum rate that they want to grant and update the information in the resource management cells. Finally, the destination reflects the resource management cells back to the sender. The closed loop resource management of ATM ABR is thus similar to the network-controlled congestion control schemes presented in this section. However, a key difference is that an ATM switch exactly knows the number of flows. This significantly simplifies the admission control algorithms and also enables variants that cannot be realized if the number of flows is not known.

4.7 Functional comparison of fast startup schemes

4.7.1 Systematic comparison of Quick-Start, XCP, and RCP

4.7.1.1 Protocol features

Quick-Start, XCP, and RCP are three congestion control schemes that all require additional signaling between endsystems and the network. They all have in common that they can *discover additional available bandwidth on the path rather quickly*, unlike end-to-end congestion control schemes, which must be conservative.

Yet, the design philosophy and protocol mechanisms are different. A systematic analysis of the design choices and different realizations can be found in Table 4.9. For the sake of completeness, ECN is also included, since it is a network-assisted congestion control scheme that is already partly used in the Internet. According to Table 4.9, the Quick-Start protocol is an evolutionary solution with a complexity between the simple ECN and the disruptive network-controlled congestion control schemes that are not downwards compatible. Its main difference compared to RCP and XCP is that it is designed for sporadic usage, i. e., the signaling is included in few packets only. Table 4.9 also reveals that the protocol specifications of XCP and RCP lack solutions for several problems that are addressed by the ECN and Quick-Start protocols. In the following sections, these problems are discussed in detail.

4.7.1.2 Common challenges of network-supported congestion control

Compared to today's end-to-end congestion control, network support would be a fundamental change that affects the Internet architecture in its core. Network-supported congestion control raises issues that have not been completely solved so far [172]:

Performance and robustness: Congestion control is subject to some trade-offs: On the one hand, it must allow high link utilizations and fair resource sharing, but, on the other hand, the

Table 4.9: Comparison of network-supported congestion control schemes

	TCP with ECN	**Quick-Start TCP**	**XCP**	**RCP**
Scope	TCP extension	TCP extension	New cong. cont.	New cong. cont.
Design target	Avoid packet loss	Fast startup	High utilization and fairness, no packet loss	Fast startup and small flow completion time
Feedback granularity	Binary	16 steps	Fine grained	Fine grained
Maximum frequency	Once per RTT	During conn. setup or after idle periods	Per packet	Per packet
Returned metric	Occurrence of congestion	Available bandwidth	Allowed increase of cong. window	Allowed sending rate
Resource sharing	N/A	Bandwidth pooling (target algorithm)	Bandwidth pooling	Oversubscription
Indication/notification	IP header	New IP option	New shim layer	New shim layer
Feedback	TCP header	New TCP option	New shim layer	New shim layer
Overhead in packets	0 B	8 B (if used)	20 B	12 B
Requires support by all routers along the path	No	Yes	Yes	Yes
Router support detection	Not required	TTL difference	Not defined	Not defined
Information required in routers	Queue length (AQM)	Order of magnitude of the link capacity	Link capacity and queue length	Link capacity and queue length
Interworking with TCP Reno	Simple if AQM is used	Simple due to fallback mechanisms	Requires traffic separation	Requires traffic separation
Malicious receiver/router protection	Optional (ECN nonce)	QS nonce	None	None

algorithms must also be robust during congestion phases. Network support can help to improve performance, but it can also result in additional complexity and more control loops. This requires a careful design of the algorithms in order to ensure stability and to avoid oscillations. A further challenge is the fact that information may be imprecise or erroneous. For instance, severe congestion can delay feedback signals. Also, in-network measurement of parameters such as RTTs or data rates may contain estimation errors. A feedback signal that is returned by end-to-end echoing has an inherent delay of one RTT. Neither the endsystem nor network components can predict how the traffic will change until the feedback becomes effective. It is an open research question how much network components can indeed improve performance without damaging or impacting end-to-end mechanisms that are already in place.

Information acquisition: In order to support congestion control, network components have to obtain at least a subset of the information in the following list. Obtaining this information may be a complex task and require additional interfaces or protocols.

1. *Capacity of links*: Link characteristics depend on the realization of lower protocol layers. Routers operating at IP layer do not necessarily know the link layer network topology and link capacities all the way to the next IP hop. The capacity is also not constant if there are shared multi-access links or links that have a variable capacity, such as wireless links or bandwidth-on-demand links. Depending on the network technology, there can be queues or bottlenecks that are not directly visible at the IP layer. Difficulties also arise due to tunnels or traffic engineering mechanisms below IP, e. g., by MPLS. In this case it may be required to coordinate the feedback in the "inner" IP header with information in the

"outer" header or label. The information about link characteristics could be determined by cross-layer information exchange, but this requires interfaces that are specific to the link layer technology. An alternative could be online measurements, but this can cause significant additional network overhead. There are also difficulties with tunnels.

2. *Traffic carried over links*: Accurate online measurement of data rates is challenging when traffic is bursty. For instance, measuring a "current link load" requires defining the right measurement interval/sampling interval.
3. *Internal buffer statistics*: Some proposals use buffer statistics such as a virtual queue length to trigger feedback. However, network components can include multiple distributed buffer stages that make it difficult to obtain such metrics.

Deployment: Schemes that require support by all network components on a path have bad incremental deployment properties. In particular, the incentive to add additional complexity to core routers is rather low, as core networks are very unlikely to become congested. This results in a typical "chicken-egg" problem.

Complexity in network components: Even if many solutions do not require per-flow state, they require additional processing in network components. Modern high-end router designs have a pure hardware data path that cannot execute complicated code per packet. Routers are optimized to process the regular IP header in the *fast path*. Unknown headers or headers with IP options are processed in the *slow path* at much lower speed. The additional processing of explicit feedback signals can therefore affect the scalability and increase the end-to-end latencies. Furthermore, additional processing efforts could expose routers to new kinds of DoS attacks.

Middleboxes: Today's IP networks include many middleboxes, such as NAT and firewalls, that may prevent the usage of transport protocols other than TCP and UDP and other new protocol mechanisms.

Security: Congestion control must operate in untrusted environments, where senders, receivers, and network components may be misbehaving. Any information exchange may create additional vulnerabilities and could offer opportunities for new attack vectors.

Multi-domain operation: Many autonomous systems only exchange very limited amount of information about their internal state (*topology hiding principle*), as such information is considered to be highly sensitive in environments with limited trust. Explicit signaling can reveal information about the internal state and dimensioning of networks that may not be visible without network support.

4.7.1.3 Issues specific to Quick-Start

There are also several challenges that are specific to the Quick-Start mechanism:

Application support: The interaction of Quick-Start with applications has hardly been studied so far, for instance, when to trigger Quick-Start requests, and how to determine the data rate to request for. Requests could for instance be triggered by the applications, but this would require modifications at the application layer. Another open research question is how to determine the data rate that Quick-Start shall request for. Both aspects are addressed in Section 5.1.1.2.

Usage of IP options: The processing IP options is non-mandatory. As a result, many routers, hosts, and middleboxes simply drop packets with unknown IP options. Measurements [150] show that a connection is not established with a probability of over 70 % if an unknown IP option is included in the SYN segment. In contrast, unknown TCP options cause only rarely problems. Other measurements have shown that packets with IP options experience higher end-to-end delays probably caused by slow path processing in routers. New IP and TCP options cannot be handled by the hardware-based TCP offload engines on modern network interface cards. Also, the TCP option space has a maximum length of 40 B and is thus scarce. In SYN segments, over 20 B are already used by TCP options in state-of-the-art TCP stacks.

Abuse: Both routers and hosts could try to report data rates that are larger then the actually available bandwidth. The QS nonce provides some protection mechanisms against cheating routers and receivers. However, QS is vulnerable to DoS attacks along two vectors: First, QS requests increase the router processing load, which could be prevented by enforcing an upper rate limit. Second, endsystems can send many arbitrarily large bogus QS requests, thus reserving the bandwidth and preventing the use of QS by other flows if bandwidth pooling is used. This problem does not exist if the approval control oversubscribes resources ("optimistic algorithm"). It could also be reduced by approval control with per-flow state [192, 191] or by policers similar to re-ECN (cf. Section 4.2.5.2).

4.7.1.4 Unsolved problems of the network-controlled schemes

Schemes such as XCP or RCP face numerous further challenges:

Support in all queues: The control algorithms of XCP, RCP, and other network-controlled schemes assume that every queue on the path supports the protocol. It is supposed that the controller has not only perfect knowledge of the link capacity, but also that there is only a single buffer in front of the link, which has a known queue size. This contradicts the design of high-end routers that rarely have one queue only, but instead consist of many cascaded or parallel queues. Also, the queues are typically located at egress line cards and the queue state is thus not easily accessible from ingress line cards, where most packet header processing is performed.

Instability: Both XCP and RCP can become unstable in certain network topologies with heterogeneous delays, if flows with long RTTs share a bottleneck with many flows having a short RTT (see, e. g., [256, 102]): If the control interval is then set to the average RTT, the control interval may be significantly shorter than the feedback loop of the flows that are bottlenecked at the queue. A mitigation is to set the control interval to the maximum RTT, instead of the average. However, then the convergence speed is much slower and the protocols become sensitive to single flows that have a large RTT or maliciously advertise a large RTT. It is an inherent problem of XCP and RCP that the control loop delay must be of the order of the RTT of the flows, which is not possible if the RTTs vary over several orders of magnitude.

Coexistence with other congestion control schemes: The congestion control design assumes that all traffic uses the proposed mechanisms. There is no simple solution how links can be shared with traffic that uses the TCP congestion control. Coexistence is likely to require separate queues and traffic isolation of the different congestion control schemes.

Incomplete specification: Network-controlled congestion control schemes such as XCP or RCP only provide a congestion control framework. As they are not a complete transport protocol, many functions must be realized on top of it. Yet, this integration is not completely described so far. For instance, the reaction to lost packets is unclear and it is assumed that it will somehow trigger a transport layer congestion control. Further important protocol functions, such as how to detect whether a path is able to support the new mechanism, are also not specified.

Security: Unlike ECN and Quick-Start, the proposed network-controlled congestion control mechanisms do not include protection mechanisms against malicious senders and receivers or man-in-the-middle attacks. A single attacker can use the protocol mechanisms to affect the resource management of all flows passing through a link. While some protection could be possible e. g. by adapting the protection mechanism of Quick-Start, this would require fundamental design changes of the protocols. In its current form, XCP, RCP, and also other related mechanisms cannot be used in untrusted environments.

In addition to these general challenges, there are also issues that are specific to each protocol: As already mentioned, XCP penalizes short flows. It is also very sensitive to endsystems that do not use their allowed bandwidth [255, 158]. RCP is vulnerable to flash crowds and may require huge buffers in certain scenarios [102], even if a recent theoretical analysis [124] shows that buffer sizes of the order of 10 % percent of the BDP may be sufficient in many cases.

4.7.1.5 Assessment and conclusions

Quick-Start has fewer open issues than other new congestion control mechanisms with per-packet feedback. In particular, it can easily coexist with other TCP traffic. Still, as some of these issues require further research, [RFC 4782] concludes that initial deployment of Quick-Start should be limited to controlled and trusted environments such as centrally administrated intranets, dedicated network for scientific computing, closed mobile networks, or satellite links. In these scenarios there are incentives to deploy new network components that include the required enhancements. A further option to get new functions in routers is a combined deployment together with other new mechanisms in the IP layer, for instance, new Internet routing schemes (cf. Section 2.4.3). However, at the time of writing this thesis, it is unclear whether such deployments are likely to occur within short time scales.

4.7.2 Other published comparative studies and related work

4.7.2.1 Simulation studies

The performance of fast startup mechanisms has already been evaluated in existing work. Concerning end-to-end schemes, the inventors of most of the schemes surveyed in Section 4.4 have publish simulation and/or measurement studies analyzing their proposed algorithms. An exception is Jump-Start: The initial evaluation in [136] is more of illustrative type and not comprehensive. It finds scenarios where Jump-Start improves performance. But in particular for medium loaded links Jump-Start has a worse performance than the Slow-Start. The drop rate is, as to be expected, higher. These studies are performed by simulation only and do not consider recent advancements in the TCP stacks.

Sarolahti *et al.* published a well-rounded evaluation of Quick-Start, also using simulations [192, 191]. The simulation results show that the Quick-Start extension can significantly enhance the TCP performance over paths with a high bandwidth-delay product. The transfer times of moderate-sized files can be improved by several hundred percent. According to these published simulation results, the overall utilization and aggregate drop rates for Web-like traffic are largely independent of whether or not Quick-Start is used, since Quick-Start is only used when the network is underutilized. The method for estimating the link utilization does not significantly affect the approval rate of QS requests, but it affects the number of failures, i. e., packet losses during the rate pacing phase. As to be expected, the "peak utilization" estimator is more conservative than the EWMA estimator. In addition to avoiding the Slow-Start, the Quick-Start mechanism has also been found to be useful in the middle of data transfers, e. g., after longer idle periods, or

Table 4.10: Reported implementations of network-supported congestion control schemes

Mechanism	Operation system	Hardware
XCP	FreeBSD, Linux 2.6 [255, 158, 102]	Network processor (partially)
RCP	Linux 2.6 [58, 102]	FPGA [58]
MaxNet	Linux 2.6 [225]	–
JetMax	Linux 2.6 [256, 102]	–
Quick-Start	Linux 2.6 [204]	Network processor [84]

after vertical handovers from narrowband to broadband links [190]. Another short study [193] analyzes the performance of Quick-Start over satellite links. It confirms significant benefits for the startup behavior of streaming traffic with TFRC.

Katabi *et al.* demonstrated by extensive simulations that XCP can achieve a high link utilization and good fairness, that the buffer occupancy is small, and that there are almost no packet drops [110]. Yet, it has also been shown that XCP can be more conservative in giving bandwidth to flows than TCP, particularly to new flows [56, 57]. XCP gradually reduces the window size of existing flows and increases the window size of new flows, making sure that the bottleneck is never oversubscribed. Short-lived flows can terminate before they achieve their fair rate. Therefore, the ramp-up speed can be smaller than the one of the TCP Reno Slow-Start, and XCP is rather ineffective for short-lived flows. Due to Little's law, this also means that there are more active flows. RCP, which should overcome this problem, is also extensively evaluated by simulations [57]. XCP and RCP have also been compared by simulation studies [57, 256] and also by measurements [102]. The author was also involved in a simulation-based comparison study of both protocols [180, 181].

Despite all this existing work, a performance comparison of end-to-end fast startup mechanisms and Quick-Start has never been published so far. Furthermore, Quick-Start and network-controlled congestion control schemes have also never been compared against each other; even though future research in this field is recommended [192]. The studies in the following section complements this existing work and fill the aforementioned gaps.

4.7.2.2 Available implementations

Simulation studies are not considered sufficient for the evaluation of new congestion control protocols. In addition to the fundamental shortcomings of simulation as method, which is explained in Section 3.3.4.1, there are also two specific problems of simulating new congestion control protocols: First, simulation models make idealistic assumptions about timing and the packet scheduling with arbitrary inter-packet delays, which cannot be realized in practice. Second, implementation issues and the complexity cannot be identified without a real implementation. This is why there have been significant efforts to implement the presented congestion control schemes in real network stacks, in particular using the Linux TCP/IP stack. The well-known implementation efforts are documented in Table 4.10, which also includes the work on Quick-Start of the author. Selected schemes have also been implemented with more hardware-support, using either a network processor or a *Field Programmable Gate Array* (FPGA). However, appart from the work of Hauger *et al.* [84], the implementation complexity of the different schemes has hardly been considered thoroughly.

5 Application integration and implementation issues and solutions

The practical realization of fast startup congestion control raises several functional issues that are not directly related to performance and the ability to deal with congestion. This chapter addresses two aspects that are not comprehensively addressed in other work: The first section analyzes how the new congestion control schemes interact with applications. It discusses the benefit of additional *cross-layer interfaces* between applications, the TCP/IP protocol stack, and also the link layer. The second section investigates interactions between fast startup congestion control and other TCP mechanisms, in particular the flow control. These issues have originally been identified by the author [202]. In addition, the actual realization of several fast startup congestion schemes is analyzed. This part of the chapter quantifies the implementation complexity and summarizes important lessons learned that have been identified as part of the research reported in this thesis. This section proves that fast startup congestion control can be realized by lightweight mechanisms. The results presented in this chapter are novel contributions beyond existing work. Unless otherwise stated, proof-of-concept implementations show that the proposed mechanisms indeed work in practice.

5.1 Solutions for the interface design

5.1.1 Application interfaces

5.1.1.1 Challenges in fast startup configuration

It is possible to realize fast startup congestion control as an extension of the TCP/IP protocol suite only. However, additional interfaces both towards applications and lower protocol layers make a lot of sense. As discussed in Section 4.3.2.2, such interfaces could provide information about application requirements. While most existing TCP algorithms are completely independent of the applications and their communication patterns, the usefulness of fast startup congestion control schemes is application dependent: An application can only significantly benefit from a fast startup (1) if it is delay-sensitive and (2) if the Slow-Start introduces a significant amount of additional delay. The needless use of a fast startup also comes at some cost: If an end-to-end mechanism is used, the risk of packet loss and congestion is increased. In case of an network-assisted scheme, like Quick-Start, unnecessary requests or requests for an overly large rate result in overhead and may waste bandwidth, too.

An activation of fast startups only for selected applications requires additional control functions: The information associated with a TCP connection in the existing network stacks is typically not sufficient in order to decide whether to activate a fast startup scheme. A sophisticated usage therefore either requires an explicit activation by an entity in the user space using an additional

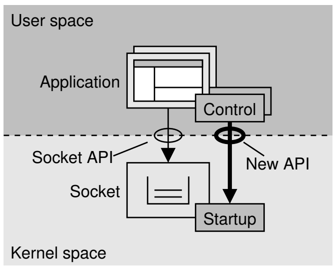


Figure 5.1: Explicit fast startup control

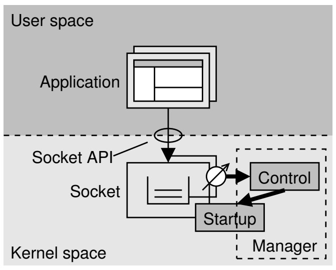


Figure 5.2: Implicit fast startup control

interface as shown in Figure 5.1, or, alternatively, additional intelligence inside the network stack. In the latter case, a *fast startup manager* inside the stack performs the control (cf. Figure 5.2). Both solutions have advantages and drawbacks: An additional interface to the user space offers more possibilities for control. It also avoids the problem that stack-internal heuristics cannot easily determine application requirements, and it maintains a clean architectural separation between application and network functions. However, the interface must be known by the applications and it must be stable and technology-independent, e. g., independent of the operating system. Instead of the application, a *congestion manager* [RFC 3124] could use this new interface, too. But this architecture would require an additional entity in the user space, and additional interfaces between the congestion manager and the applications.[1] In contrast, a stack-internal fast startup manager does not mandate any additional interface to applications and is therefore simpler to realize. It can also handle existing applications that do not use new interfaces, and it could even try to classify applications implicitly according to their traffic patterns. The question whether the service requirements of applications are implicitly or explicitly obtained is an inherent trade-off, which is already discussed by Shenker [212].

5.1.1.2 Parametrization

All fast startup schemes have parameters that must be configured. In particular, Quick-Start and similar mechanisms (e. g., Mega-Start) require that the sender selects an initial data rate. This *initial rate selection problem* is non-trivial without additional context information about the application and/or the environment. There are five possible sources from which relevant information could be obtained:

1. *Application*: An application may know what data rate would be useful. In some cases such information is already known at compile time, e. g., in case of multimedia streaming with codecs that have a known average or peak rate. Alternatively, it can be a run-time parameter that must be configured by the user, similar to other network-related parameters such as the use of an HTTP proxy. Furthermore, an application may automatically determine an initial rate from knowledge or predictions of the communication characteristics and performance requirements. This solution is the recommended one in this work.
2. *System configuration*: The administrator of the endsystem could set global configuration parameters. For instance, a parameter could specify the capacity of the local interface or the capacity of the Internet access, and this value could be used as initial rate.

[1]Alternatively, a congestion manager could intercept the sockets interface function calls, e. g., by *dynamic-link library injection*, and then activate the fast startup congestion control on behalf of the applications. This solution would not require modifications of the applications and is equivalent to a stack-internal control mechanism.

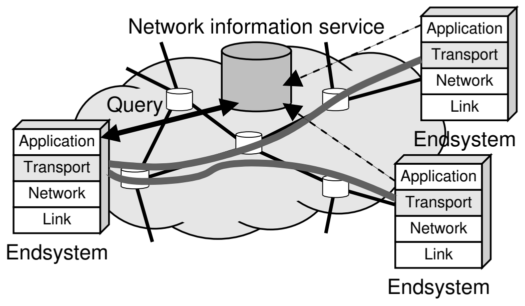


Figure 5.3: Querying fast startup parameters from a network information service

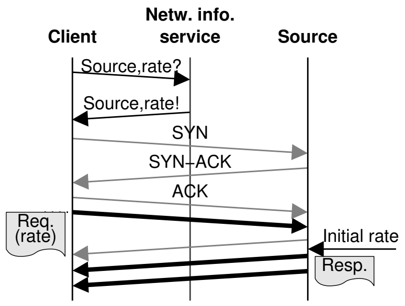


Figure 5.4: Using signaling in order to obtain a rate recommendation

3. *Stack-internal heuristics*: A *fast startup manager* inside the stack can monitor the application communication behavior and path characteristics and learn from this. For a single connection, an obvious option is to measure the available bandwidth on the path during data transfers and use this rate after longer idle times, when the Congestion Window has been reduced. In case of Quick-Start, if there has been a previous successful request, also the last granted rate could be used when a new request is issued. A fast startup manager could also try to classify applications by observing their communication characteristics, e. g., by monitoring the amount of queued data in the socket as shown in Figure 5.2. Finally, there could be information sharing between connections as introduced in Section 4.4.1.2, or a full-featured *congestion manager*.
4. *Network interface*: Reasonable choices for the initial rate could be made by knowing the local link capacity [RFC 4782]. Such network characteristics could also be obtained by cross-layer information exchange with the network interface (see Figure 4.15).
5. *New network information service*: Recommendations for initial sending rates could also be obtained from centralized entities by a signaling control plane. In ongoing research and standardization work, a new network information service is developed that provides guidance to applications how to optimize their traffic [4, 251]. The main motivation is to optimize the topology of P2P systems by providing access to information about the internal network topology. As illustrated in Figure 5.3, this information could be provided by a query/response protocol to servers that are operated by Internet service providers. But this service could also expose other information, such as the *capacity of the last mile link*. If such a service was available in future, the returned capacity information could be used as a initial sending rate during flow startups. Figure 5.4 shows one possible realization of such a flow startup with a recommended initial sending rate. As the realization of this network information service is still a research issue, its usage in combination with fast startup congestion control is not further detailed in this thesis.

5.1.1.3 Proposed new application interface

The sockets interface [1003.1] is the most widely used interface between the TCP/IP stack and applications. The transport protocol services are accessed in the same way as reading or writing from a file. The weakness of this simple and abstract API is that it hides all network-specific information (*context information*) and isolates the application logic from the transport protocol congestion control (see Figure 4.15). The current TCP/IP stacks hardly offer control interfaces

Table 5.1: Proposed fast startup application interface extensions

Functions	**Suggested values**
TCP_APPLICATION_TYPE	RESPONSIVE/STANDARD/LOWPRIORITY
TCP_INITIAL_RATE	Rate [bit/s]
Potential additional functions	**Suggested values**
TCP_STABILIZE_THROUGHPUT	True/false
TCP_DATA_VOLUME	Expected transfer volume [B]
TCP_DEADLINE	Desired delivery deadline [ms]

to applications apart from the standardized `TCP_NODELAY` socket option [1003.1]. In particular, there is no possibility to affect the sending rate determined by the congestion control. It has been argued that further optional, generic, and technology-independent interfaces are needed [61], but the design of such interfaces is still an open issue. The availability of fast startup congestion control may improve the acceptance of such new interfaces.

All fast startup mechanisms developed in this work support an implicit activation by heuristics. In some cases there are also further control possibilities: Several *new control primitives* would enable the control of fast startup congestion control schemes by applications. This thesis proposes a number of new "set" functions that are listed in Table 5.1. They are independent of the realization of the fast startup mechanism.

The first proposed function call allows an application to characterize its communication requirements according to three classes, which represent the main TCP usage scenarios:

1. *Responsive*: Activation of fast startup congestion control
2. *Standard*: Usage of the standard TCP congestion control (default)
3. *Low priority*: Usage of a less than best effort congestion control

The default class for elastic applications is *standard*; it results in the usage of the Slow-Start algorithm. A broadband interactive application that wants to apply fast startup congestion control can set the type of a connection to *responsive*. In addition to this, *low priority* class makes sense for background transport (cf. Section 4.2.3.2), but the realization of this class is not further studied in this work. These three classes overcome the problem of designing *one* congestion control for all kinds of applications, while still keeping the number of classes small. The provisioning of a faster application class is well aligned with the arguments of Briscoe that suggests to speed up Web applications compared to bulk data P2P traffic [35]. In a DiffServ-enabled network, the application classes could also be mapped to the corresponding DiffServ classes listed in Table 2.1, if DiffServ is supported on the path. For example, the *responsive* application class could be mapped to the *low-latency* DSCP. A further, unexplored possibility would be to control the flow startup aggressiveness by a numerical weight parameter.

The second proposed function call allows applications to inform the network stack about a desired reasonable rate. It implements the interface that has been introduced in the previous subsection. As there is no possibility for resource reservation, the value is a recommendation only. The actual speed of the flow startup may of course be smaller.

Further useful function calls could characterize whether the requested rate is expected to be sufficient (e. g., if it is the peak rate), or whether higher data rates would be beneficial. This information may be important in order to decide how to handle the SST after a fast startup

```
int initial_rate = 10000000; /* 10 Mbit/s */
setsockopt(socket, SOL_TCP, TCP_INITIAL_RATE, &initial_rate, sizeof(int));
```

Figure 5.5: Explicit application interface example: The source code in the C programming language shows how an application could activate a fast startup by a new socket option.

phase, as discussed later in Section 6.3.3.1. Table 5.1 also lists further possible function calls that could be used as well. For example, it can make sense to inform the network stack about the total amount of expected data, since applications do not write data to the socket in a single call [192].[2] A further option would be a function by which an application could announce the *future* availability of data, potentially even with an estimation of the delay until the data will be available. Such an announcement could be beneficial in combination with a signaling mechanism like Quick-Start. In theory, an early announcement could help to start the signaling right before the data actually arrives. In practice, however, this is difficult since a Quick-Start request can only be signaled in <SYN> or data segments and therefore cannot be sent in advance. This is why only the first two inferfaces are specifically studied in this work.

All proposed interface extensions are optional, i. e., if an application does not use them, the standard TCP congestion control will be used. The interface can easily be realized by additional socket options. The usage by applications is very simple. For instance, Figure 5.5 illustrates how an application can request for an initial sending rate with two lines of code only, which can easily be added to the source code of TCP-based applications, e. g., Web servers [223, 204].

5.1.1.4 Application integration

A non-trivial question is *when* a source can provide information about its desired rate. This is of particular importance for schemes that need one RTT for signaling, such as Quick-Start. Figures 5.6 and 5.7 compare two different cases how an application could activate Quick-Start: It could either enable Quick-Start before the connection is set up (*early activation*). Then the Quick-Start request can be piggybacked during the three-way handshake. If it is approved, a data transfer could immediately start if application data is available within one RTT. However, in a request-response protocol such as HTTP, the server does not necessarily know the size of the requested object during TCP handshake [192]. This makes it difficult to decide whether to

[2]An alternative in the Linux network stack is the `TCP_CORK` socket option.

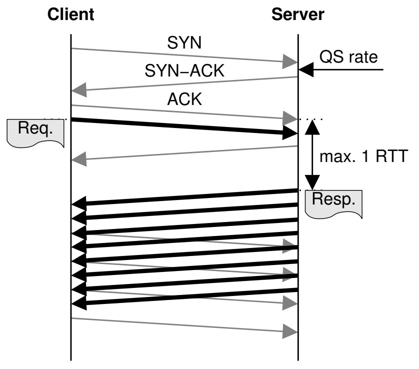


Figure 5.6: Early activation of Quick-Start

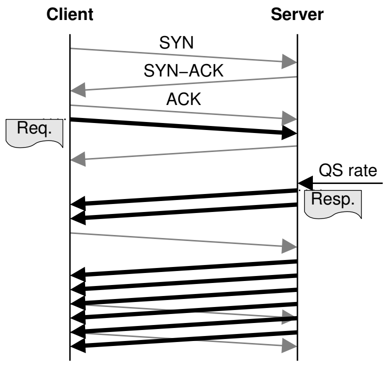


Figure 5.7: Late activation of Quick-Start

```
scharf@tcp1:~$ sudo ifconfig eth1 capacity 10000000
scharf@tcp1:~$ ifconfig eth1
eth1      Link encap:Ethernet HWaddr 00:80:C8:F6:91:04
          inet addr:192.168.0.1 Bcast:192.168.0.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
          Capacity:10000000 bit/s
          Interrupt:17 Base address:0xd800
```

Figure 5.8: Example for the manual command line configuration of the interface capacity for the router functions of network-supported congestion control. The “capacity” parameter is a new interface extension. In the example, the maximum data rate available to Quick-Start on an Ethernet link is set to 10 Mbit/s, which is a conservative setting.

use a fast startup, or not. One possible remedy would be to use dedicated sockets for the data transfers for which a fast startup is desired.

An alternative is to realize the rate request only when the request has already been processed and when the response arrives, i. e., when a more informed decision can be realized. This *late activation* is depicted in Figure 5.7. In case of Quick-Start, the late activation requires that the Quick-Start is piggybacked on the first data packets. This means that the first packets are sent as allowed by TCP's initial window, and that there is no fast startup during the first RTT. After one RTT, subsequent packets are then sent with the approved rate.

Obviously, a fast restart can only use the late activation. In general, the late activation is required for all fast startup schemes that require a signaling handshake that includes application requirements. End-to-end fast startup mechanisms do not suffer from the delays caused by the *late activation*. This is an important advantage compared to network-supported schemes.

5.1.2 Network interfaces

If a fast startup scheme requires knowledge about the network interface capacity, additional interfaces between the network and/or transport layer are required, too. As discussed in Section 4.7.1.2, this is a key requirement of network-supported congestion control schemes. It depends on the link layer technology whether the link capacity is known, and whether and how this information can be made available to the TCP/IP stack. For example, the operating system on a PC with Ethernet links could automatically determine the current interface link speed from the corresponding device driver. Due to auto-negotiation mechanisms, this information must be updated periodically. On multi-access links the problem is more challenging, and a cross-layer information exchange may be required if the exact link capacity must be known. One exception is Quick-Start, which also works if the assumed link capacity is smaller than the actual value [RFC 4782]. Thus, the assumed capacity can be set to a reasonable lower bound.

This work suggests to extend the interface configuration tools in order to configure the capacity parameter. Figure 5.8 shows an example for such a command line tool extension that has been developed as part of the Linux Quick-Start implementation. The interface presented in Figure 5.8 is simple and straightforward to use, as only one additional parameter has to be configured per link. A network-controlled congestion control scheme that requires a very precise knowledge about the available bandwidth could require a more sophisticated interface.

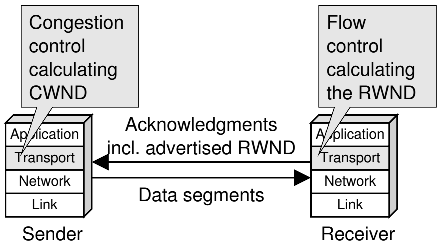

Congestion control calculating CWND
Flow control calculating the RWND
Application
Transport
Network
Link
Acknowledgments incl. advertised RWND
Data segments
Sender
Receiver

Figure 5.9: Illustration of the difference of congestion control vs. flow control

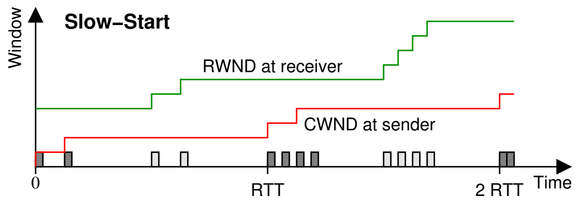


Figure 5.10: Illustration of receive window auto-tuning that assumes a Slow-Start

5.2 Proposed mechanisms to avoid interactions with flow control

5.2.1 State-of-the-art receive window auto-tuning

TCP realizes congestion control and flow control. In principle, both mechanisms are independent. The former is a sender-side closed loop control mechanism that determines the available capacity on the path. The latter is a receiver-driven open loop control that informs the sender about the available receive buffer at the receiver, which is announced by the *advertised receive window* as depicted in Figure 5.9. The receive window avoids that the sender sends data that will be dropped at the receiver. As a consequence, both congestion and flow control have similar objectives, namely avoiding waste of network resources, and they may interfere.

A receiver must consider two different constraints when determining the size of the advertised RWND: On the one hand, the available memory must be subdivided among the existing TCP connections. On the other hand, the available bandwidth can only be used if the advertised RWND is of the order of the BDP of the path. It has been argued a long time ago that TCP receivers should advertise large windows no less than twice the BDP [210], so that the path is efficiently used and out-of-order segments can be stored, too. Receivers can even oversubscribe their buffer space because the whole buffer is hardly ever needed [210].

In modern operating systems, the receive buffer is not statically allocated. *Automatic buffer tuning* dynamically determines the socket buffer sizes based on available memory and empirical measurements: First, the receiver should measure the application processing capacity. If enough buffer space is available, it should advertise at least an RWND equivalent to approximately twice the amount of data read by the application in one RTT. Second, the receiver should estimate the BDP of the path (or the sender's CWND) and try to advertise a receive window of twice that value. This so-called *dynamic right-sizing* [69] ensures that the data transport is not constrained by flow control as long as the bottleneck is not in the receiver. Automatic buffer tuning is typically also implemented on the sender side and then known as *auto-tuning* [210]. For the Linux operating systems, the complex details of the algorithms used for automatic buffer tuning and receive window advertisements are described in [218]. According to Section 4.2.4, automatic buffer tuning is also used in other important modern TCP/IP stacks.

5.2.2 Possible interactions with fast startup congestion control

The usage of a fast startup significantly changes the TCP behavior during connection setup, since a sender can use a large CWND immediately after the connection setup. This results in two potential *interactions between the TCP flow control and a fast startup congestion control*

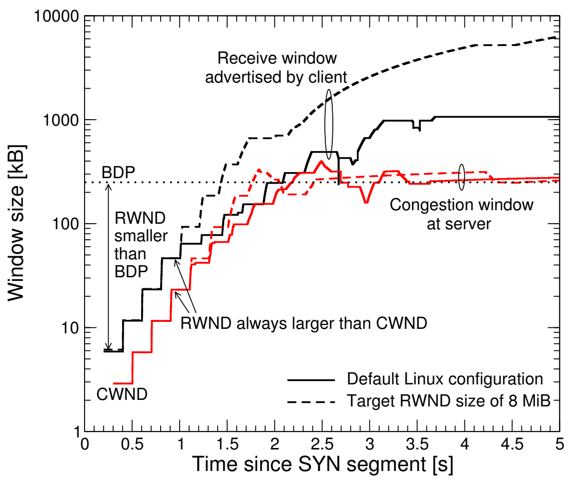


Figure 5.11: Example for the receive window auto-tuning used by Linux

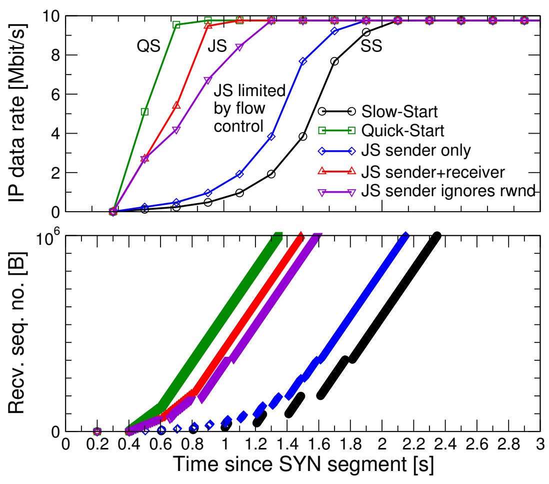


Figure 5.12: Illustration of the interaction between a fast startup and flow control

scheme, which are described in the following. Both problems as well as corresponding solutions are also comprehensively specified in [202].

First, the receiver might not allocate a sufficiently large buffer space after connection setup, or it may advertise a small receive window implicitly assuming the Slow-Start behavior on the sender side. This effect is illustrated in Figure 5.10. Existing automatic buffer tuning mechanisms initially advertise a rather small receive window. The more data arrives, the more buffer space is advertised. This behavior is reasonable if the sender uses the standard Slow-Start and indeed starts with a small congestion window [69]. However, when a fast startup shall be used, the receiver must be ready to buffer a large amount of data immediately after the connection setup, which requires a *modified buffer allocation strategy*.

Figure 5.11 demonstrates this problem using the example of the Linux stack: It shows both the CWND and the advertised RWND of a TCP connection over a 10 Mbit/s link with a minimum RTT of 200 ms (see Section 6.1.1.1 for further details on the simulation scenario). In the graph of the CWND, one can clearly observe the initial Slow-Start with an initial window of $w = 2$ MSS. The RWND announced by the Linux receiver is approximately two times as large as the CWND and thus follows a Slow-Start as well. This virtually prevents a fast startup: Even if the sender increased its CWND faster, the transfer would hardly be speeded up, as the amount of outstanding data would then be limited by the small RWND.

This interaction of TCP flow control and fast startups is also confirmed in Figure 5.12. It covers two different configurations: If both sender and receiver support fast startup congestion control, the flow control implementation can be modified in order to announce a sufficiently large window. Figure 5.12 confirms that when both endsystems use Quick-Start or Jump-Start, the fast startup is realized as expected. However, if only the sender uses Jump-Start, the receiver-side flow control enforces a Slow-Start. A more detailed analysis of the differences in the startup behavior can be found later in Section 6.3.

Second, there is a specific issue with the *semantics of the receive window*: The TCP header standardized in [RFC 793] uses a 16 bit field to report the RWND size to the sender, which limits the value to 64 KiB. In order to circumvent this limitation, the *window scale* TCP extension [RFC 1323] defines a scale factor that is used to multiply the window size value in a

TCP header to obtain a 32 bit value. If enabled, the scale factor is announced during connection setup by the window scale TCP option in <SYN> and <SYN,ACK> segments. However, [RFC 1323] defines that the "Window field in a SYN (i. e., a <SYN> or <SYN,ACK>) segment itself is never scaled". This means that the maximum receive window that can be signaled to the sender in the <SYN,ACK> is 64 KiB. This significantly limits the usefulness of a fast startup after the three-way handshake in the direction of the connection initiator to the responder. After having received the <SYN,ACK>, the connection initiator cannot send more than 64 KiB before receiving an ACK, even if the fast startup congestion control scheme allows a much larger Congestion Window and if the receiver actually has much more free buffer space.

This restriction is particularly undesirable in combination with the Quick-Start mechanism [202]: First, if a sender sends the Quick-Start request in the initial <SYN> segment, and if the corresponding Quick-Start response is echoed back in the <SYN,ACK>, the receiver already knows before sending the <SYN,ACK> that a large burst of data may arrive. It can allocate sufficient buffer space, but it cannot announce it due to TCP header semantics. Second, as the amount of data sent during the Quick-Start rate pacing phase is at most 64 KiB, the CWND is set to this value after leaving the rate pacing phase [RFC 4782], even if the granted data rate would have allowed a much larger CWND.

Both the problem of receive buffer dimensioning as well as the limitations of the TCP header semantics apply to all TCP-based fast startup schemes. The possibility of a limitation by receive window auto-tuning has also been observed in other implementation work [255, 158], but solutions have not been proposed. The interaction between fast startup schemes and [RFC 1323] are also completely neglected in many related NS-2 simulation studies such as [192, 136].

5.2.3 Proposed solution and its implications

These two interactions between congestion control and flow control can be avoided by two optional changes of the TCP algorithms [202].

Optimistic buffer management: If a receiver wants to allow a sender to use a fast startup congestion control, it should advertise a sufficiently large initial receive window so that data transfers can indeed start with a high sending window. A reasonable value is the expected bandwidth-delay product of a path, if the endsystem has sufficient socket buffer space. The amount of information that a receiver can gain during a connection setup depends on the fast startup mechanism. In case of end-to-end fast startup schemes, a receiver cannot easily determine whether a sender uses fast startup congestion control, i. e., it may always have to announce a larger window.[3] If an explicit signaling scheme such as Quick-Start is used, the receiver has additional information and can announce larger receive windows only if Quick-Start requests are received. For instance, the endsystem could estimate the required buffer size as the product of the approved Quick-Start rate and the RTT, and advertise a corresponding receive window. This receive window should allow the other TCP host to fully use the approved Quick-Start request. If the RTT is unknown a worst-case RTT such as 500 ms could be assumed.

Additional acknowledgment method: The proposed solution for overcoming the limitation imposed by the window scaling is an additional acknowledgment. If necessary, the connection responder host could send a scaled receive window in a separate <ACK> segment following

[3]One solution would be a new TCP option by which the sender announces its willingness to use a fast startup. This straightforward TCP extension is left for further study.

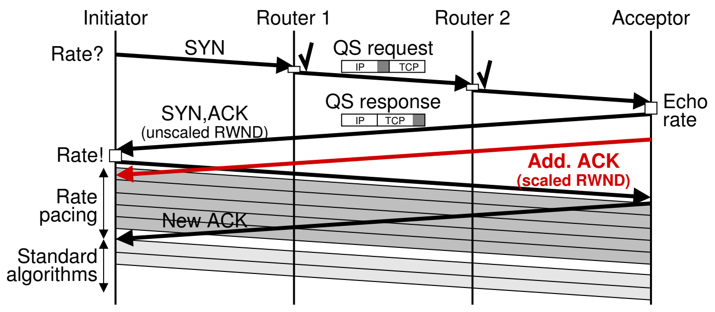


Figure 5.13: Message sequence chart of the workaround to prevent interactions between fast startup congestion control and the TCP flow control, shown for the example of Quick-Start

the <SYN,ACK> packet that announces the true scaled receive window. The resulting message flow is depicted in Figure 5.13. After having received this additional acknowledgment, the sender is aware of the true available receive buffer. There is some degree of freedom as to when to send the additional acknowledgment. The straightforward solution is to send it immediately after the <SYN,ACK> segment. But in fact it is sufficient if the sender receives this segment before reaching the limit of the unscaled receive window. As a consequence, receivers could also delay the sending of this segment for some small amount of time.

The specification of these two methods [202] comprehensively discusses their implications, deployment options, as well as security issues, in particular focusing on the Quick-Start TCP extension. Both methods are compliant with the TCP specifications [RFC 793, RFC 2581]. For standard-compliant TCP stacks, their realization should require changes in the receiver TCP implementation only. However, practical experiments have shown that sender-side modifications are also required to enable the processing of the additional ACK. In general, an empty acknowledgment shortly after a <SYN,ACK> segment is an atypical TCP communication event. Both segments could get reordered or the additional ACK may be dropped by middleboxes. In these cases, the sender only knows the unscaled receive window until the next new ACK arrives. Delaying the additional ACK could help to avoid such problems [202].

Both methods also have *security implications*. If an endsystem reserves large amounts of buffer space during the three-way handshake, this could increase the vulnerability to *SYN flooding attacks*. An attacker sending many SYN segments could try to allocate much buffer space at a host. This problem can be solved by buffer oversubscription, which works well as long as receiving applications are able to process data as fast as it is delivered by the network. The usage of an additional ACK in order to update the receive window also causes some limited additional network overhead. Since two packets are sent as response to a single <SYN> segment, an attacker could spoof its source IP address and use the ACKs for a DoS attack with an *amplification factor* of two. However, this is not a major security thread.

5.2.4 More disruptive alternative solutions

The limitation by the TCP flow control could be circumvented in several other ways, which are mentioned here for the sake of completeness. One could use a scaled receive window in <SYN> and <SYN,ACK> segments, but this would cause interworking problems with current TCP implementations. The scaled receive window could also be indicated by a new TCP option.

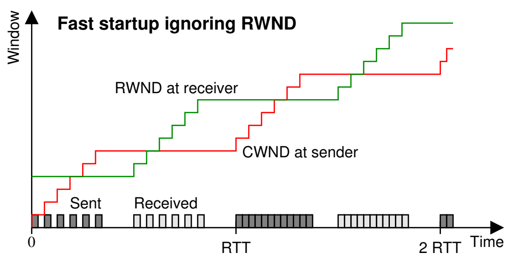


Figure 5.14: Explanation why the RWND could be ignored during the flow startup

Another, more disruptive possibility is that the sender *ignores the advertised receive window* during a fast startup. This violation of the TCP specification [RFC 793] indeed works with unmodified TCP receivers: Even if the sender sends more data than allowed by the receive window, TCP's protocol semantics are always fulfilled at the receiver side, if the receiver uses receive window auto-tuning with a receiver-side Slow-Start. As shown in Figure 5.14, the receiver increases RWND whenever new segments arrive. This means that the computed receive window at this point in time is always large enough to accommodate the arriving segments – unless the receiver runs out of memory. If a receiver does not have enough buffer space, it can simply drop the arriving packets. The reaction of the TCP congestion control then avoids a further increase of the buffer space consumption in the receiver.

Figure 5.12 shows that this disruptive mechanism could be a solution, too: An unmodified Linux-based receiver indeed accepts the packets sent by the Jump-Start scheme even though its announced RWND is small. Note that this Jump-Start implementation only violates [RFC 793] during the first RTT and complies to the receive window afterwards. As a result, the speedup is slightly smaller compared to bi-directional fast startup support. In the experiments in this thesis, it is generally assumed that both endsystems support the fast startup and that the receiver both announces a large RWND and sends an additional ACK if required. Still, Figure 5.12 shows that it is possible to obtain similar results with *sender-side modifications* only.

From a fundamental point of view, the interaction issues result from a *functional overlap* between congestion control and flow control. It is an interesting insight that, in fact, the TCP flow control is actually not a mandatory TCP mechanism, since the congestion control alone could handle receiver overload as well: Because an overloaded receiver does not accept arriving packets, the congestion control is triggered anyway. As a consequence, the author has also proposed to design a novel TCP flow control that only considers the receive window when its value is smaller than a threshold, and solely relies on the congestion control algorithms otherwise [201]. This simple modification would significantly simplify the socket buffer memory management in combination with fast startup congestion control. However, disabling the flow control also has some drawbacks, since the advertised RWND can announce additional capacity very quickly in a single ACK, which is useful if applications consume data in bursts. Further studies would be needed to study all implications of such a simplified TCP flow control.

5.3 Realization complexity and feasibility

5.3.1 Overview of the implementation work

Any new protocol mechanism raises the question whether it can be implemented in an effective and efficient way. This question is of particular importance for any proposal that adds

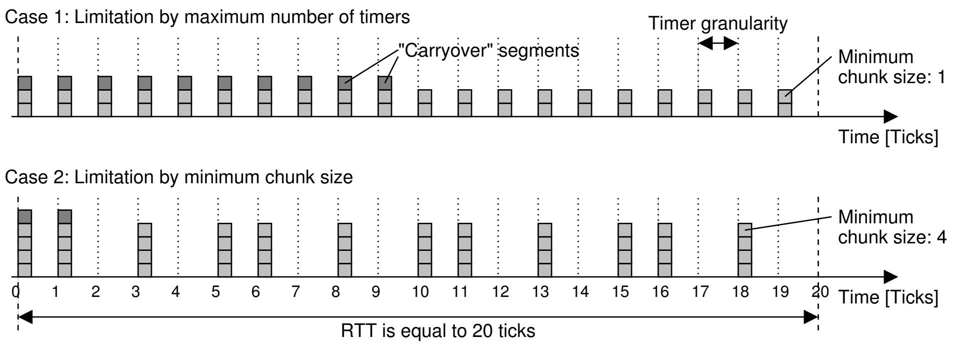


Figure 5.15: Rate pacing with a constant timer granularity. In the example a window of $W_{\text{playout}} = 50$ is played out during an RTT of $20/HZ$ with different minimum chunk sizes N_{chunk}. A larger value of N_{chunk} can reduce the number of required timers.

additional complexity to network components. This section summarizes the results of a case study that compares the implementation complexity of the Quick-Start protocol to other fast startup schemes, both concerning endsystems and network components. The realization efforts are compared to Jump-Start as an example for a pure end-to-end mechanism. Both schemes, as well as the other TCP extensions have been implemented in the Linux network stack [204, 205] in order to show that they can be realized with limited overhead. The Quick-Start and XCP processing has also been implemented in a network processor in a research cooperation with Hauger [84, 203]; the corresponding lessons learned are also briefly summarized here. The numerical performance results are presented later in Section 6.6.

5.3.2 Implementation in endsystems

5.3.2.1 Overview of the Linux implementations

The endsystem implementations are realized as modifications of the Linux network stack. Linux is an open source operating system with a highly optimized and field-proven network stack that is widely used as development platform for experimental Internet research (cf. [240]). The first implementation of Quick-Start in the Linux kernel has been realized by Strotbek [223]. The kernel patch completely implements Quick-Start for IP version 4 and TCP according to the specifications [RFC 4782, 202], i. e., both endsystem and router functions. This implementation has subsequently been extended and improved by the author [204]. The version used in the experiments in this work is based on Linux kernel version 2.6.24. The implementations of Jump-Start and other fast startup schemes modify the same kernel version.

5.3.2.2 Realization details of the rate pacing

Most proposed fast startup schemes use a rate pacing mechanism. As TCP is a window-based protocol, a standard TCP/IP stack does not use rate pacing. Rate pacing is thus one of the new mechanisms that have to be added to a protocol stack. Its realization is non-trivial because it requires timers, which are only possible with a certain granularity. In its default configuration, the Linux kernel uses an interrupt to realize kernel-internal timers, which is called $HZ = 1000\,1/\text{s}$ times per second. Thus, Linux offers a high timer granularity of 1 ms.

In the fast startup implementations, the rate pacing is realized by such an additional kernel-internal timer. Each time the timer expires, the Congestion Window W is increased by a given

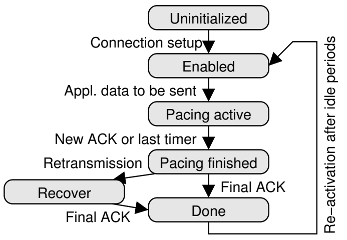


Figure 5.16: Simplified Jump-Start sender state engine

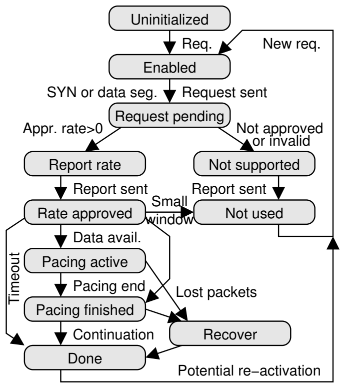


Figure 5.17: Simplified Quick-Start sender state engine

amount of segments up to target window W_{playout} (e. g., W_{qs}). The fixed timer granularity thereby affects the number of timers N_{timer}: As shown in Figure 5.15, the number of timers depends on the window increase W_{playout}, the minimum increase per timer N_{chunk} (*minimum chunk size*), and the number of *ticks* of the rate pacing duration. Pacing over one RTT corresponds to a maximum number of timers of $\lfloor \tau \cdot HZ \rfloor$. The actual number of timers is therefore

$$N_{\text{timer}} = \min\left(\lfloor W_{\text{playout}}/N_{\text{chunk}} \rfloor, \lfloor \tau \cdot HZ \rfloor\right). \tag{5.1}$$

As can be seen in Figure 5.15, the configuration parameter N_{chunk} trades off the traffic burstiness and the number of timers, i. e., the processing overhead. If the number of window increase steps $\lfloor W_{\text{playout}}/N_{\text{chunk}} \rfloor$ is large, the maximum number of timers is required, and *carryover segments* might be required [223]. The overhead of many timers can be reduced by enforcing a minimum window increase per timer $N_{\text{chunk}} > 1$. In this work, a minimum chunk size of $N_{\text{chunk}} = 3$ segments is used, which corresponds to the initial burstiness allowed by [RFC 3390]. However, there are also cases in which the timer granularity of the operating system is not sufficient for a fine-grained rate pacing (i. e., $N_{\text{timer}} = \lfloor \tau \cdot HZ \rfloor$). Then, the burst size can be significantly larger.

5.3.2.3 Case Study: Comparison of Jump-Start and Quick-Start TCP

In order to compare the overhead of a network-assisted fast startup scheme to an end-to-end solution, the following section studies the similarities and differences of the implementations of Jump-Start and Quick-Start TCP. In both cases, the sender must keep additional state information. Figure 5.16 and Figure 5.17 compare the sender's state engines in the implementations used in this work. In case of Jump-Start, the operation is rather straightforward, even though some complexity is caused by the modified error recovery procedures. Also, the socket processing workflow must be adapted in order to determine the amount of queued application data. Quick-Start requires much more states to handle the sending and reception of the IP and TCP options. [RFC 4782] also defines several abort conditions that must be taken into account.

One question that is not addressed in literature is how and when to activate the fast startup. The Quick-Start implementation used in this work supports both an explicit enabling by the application, using the socket option interface introduced in Section 5.1.1, as well as kernel-internal heuristics. They automatically issue a request for the previous rate if application data is available after a long idle time. These heuristics can also be used in the Jump-Start implementation in a similar way.

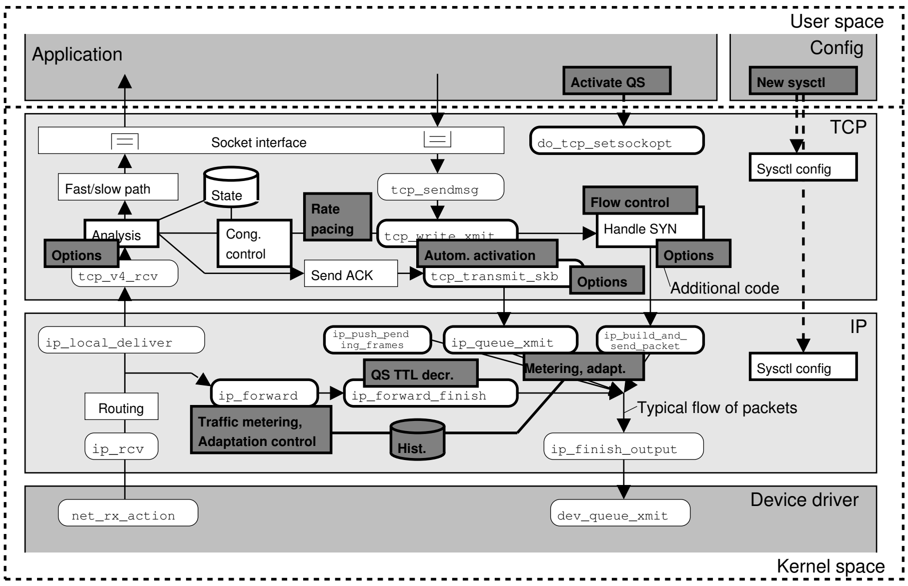


Figure 5.18: Illustration of the Quick-Start TCP implementation in the Linux network stack. The required modifications are highlighted by the gray boxes.

In order to illustrate the complexity of these implementations, Figure 5.18 gives a simplified overview of the structure of the IP and TCP layer in the Linux kernel, focusing on the packet processing functions. The gray boxes represent the main modifications of kernel code that are required to implement Quick-Start. Even though the Quick-Start mechanism is rather simple, changes are required in almost every part of the TCP implementation, and also in several IP functions. The Quick-Start IP option processing is here realized by new methods in the forwarding path of IP packets, which meter the traffic, perform the approval control, and store recently approved requests in a ring buffer as described in Section 4.5.2.3. An alternative solution would have been to use the Linux "netfilter" mechanisms as in related work [158, 255, 102]. The TCP implementation must be extended by the processing of the header options, the modification of TCP congestion and flow control, and the rate pacing mechanism. Global and interface-specific parameters can be configured by additional `sysctl` variables and `ioctl` calls, which are listed in Appendix B.3. Further details about the Quick-Start implementation can be found in [223]. Compared to this, a realization of Jump-Start is much simpler and mainly consists of a subset of the modifications in the TCP layer.

A quantitative comparison of the complexity of the Jump-Start and Quick-Start implementation is provided in Table 5.2. While these numbers are affected by some design choices, they

Table 5.2: Comparison of the implementation complexity of fast startup schemes

Criteria	**Jump-Start**	**Quick-Start**
Additional lines of code (including comments)	ca. 600	ca. 2200
Number of affected source code files	11	24
New state variables per connection in `tcp_sock`	9	19
New state variables per interface in `net_device`	–	10 or more
New configuration parameters in `/proc/sys/net/ipv4`	4	13

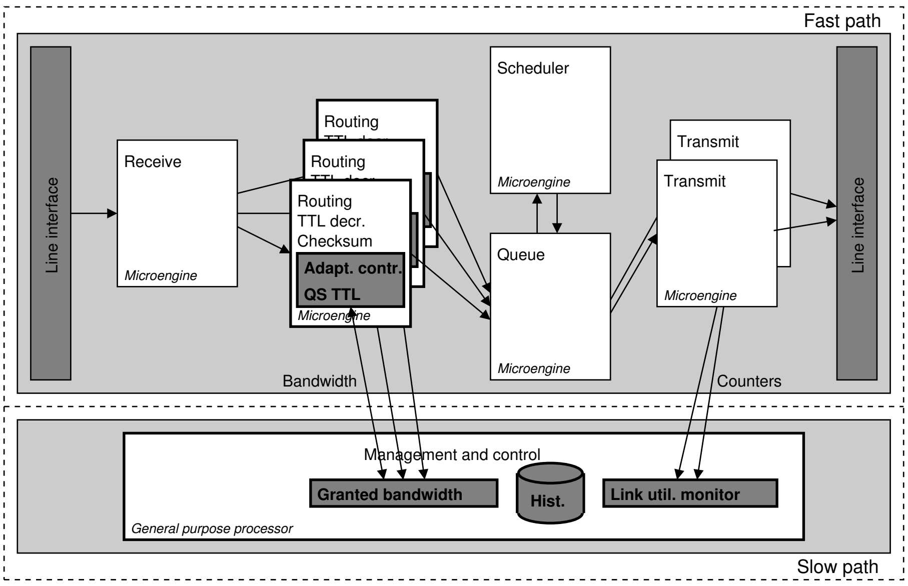


Figure 5.19: Illustration of a Quick-Start implementation in a network processor (adapted from [84, 203])

show the overhead caused by network support. Compared to the complexity of a TCP/IP stack as a whole, these numbers are rather small. For instance, the Quick-Start patch changes less than 5 % of the TCP and IPv4 code. This means that, in terms of lines of code, the Quick-Start implementation is at least one order of magnitude less complex than a user-space RSVP implementation [109]. The available Linux implementations of XCP [158, 255] or RCP [58] have a similar complexity like the presented Quick-Start implementation, even though they only address a subset of the problems solved by Quick-Start.

5.3.3 High-speed implementation in network components

Any network-supported congestion control scheme that requires additional processing in routers can only be realized if the required functions can be implemented efficiently in high-speed routers. Modern routers have a modular architecture consisting of network interface cards, an internal interconnection unit (*switch fabric*), and a *Central Processing Unit* (CPU) [14]. Packets are either processed in the *fast path* or in the *slow path*. The fast path is optimized for processing of the vast majority of packets and is typically implemented in hardware, i. e., in *Application-Specific Integrated Circuits* (ASICs) or in *Field Programmable Gate Arrays* (FPGAs). The slow path processing is performed in software and used for packet that are less time critical. One fundamental challenge for IP enhancements is that packets including IP options are typically processed in slow path [14]. This means that such packets can only be processed at a limited rate and may suffer from larger delays than packets in the fast path.

In order to prove that the router functions of network-supported congestion control can be realized in the fast path with very limited effort, Suriyajan [226] has implemented the Quick-Start and XCP router functions in an IXP 2400 *network processor*. A network processor is a programmable hardware component that is optimized for high-speed packet processing and that integrates one or more special purpose processors. The IXP network processor series comprises multiple cores for fast path processing (*microengines*) and is widely used for experimental networking research.

The resulting structure of an IP router implementation with added Quick-Start support is illustrated in Figure 5.19. The figure shows that the router functions are divided into several functional pipeline stages. The processing of the Quick-Start requests is added to the pipeline stage that performs the IP packet processing, and the management of the spare capacity on the output links is realized by the general purpose processor. A more detailed description of the implementation can be found in [226, 84]. The measurement results published in [84, 203] show that both Quick-Start processing is feasible at multiple Gbit/s line speed. Some numerical performance results are also summarized in Section 6.6.

In addition to Quick-Start, XCP is also considered in references [226, 84]. Compared to Quick-Start, XCP requires a more complex processing. The main difference is that the additional effort is required for every packet, whereas Quick-Start options can be expected to be used in only few packets. This difference is very important if packets are processed by several entities in parallel. In this case, *synchronization* of global state variables among the different microengines must be ensured. As explained by Hauger [84], a general solution for synchronization is *locking*, but locking does not scale well with parallelism. A more efficient realization of the target algorithm approval control of Quick-Start is possible by *atomic operations*. However, this method cannot be applied for the complex XCP algorithms, which can result in performance problems.

5.3.4 Lessons learned from the case study

The most important result of the case study is that fast startup schemes such as Jump-Start and Quick-Start can be implemented in a state-of-the-art network stack with a very limited additional complexity. The realization of an end-to-end scheme such as Jump-Start is mostly straightforward. For Jump-Start, the only major issue that has been identified in real-world tests is that its performance significantly depends on the question whether an application writes data as a whole to the socket buffer. If applications use several small write operations, the data transport could be speeded up by rewriting application code.

The implementation work on Quick-Start revealed several problems that are not mentioned in the protocol specifications but that are important in practice:

- *Storage of state information*: An endsystem that initiates a Quick-Start request must store several integer variables in the *TCP control block*. This state information is critical in case of *listening sockets*, i. e., if connection acceptors trigger a request in the <SYN,ACK>. When there are multiple parallel connection setups to the same listening socket, the endsystem must store the Quick-Start state information for each of them. But the network stack should actually store only the minimum amount of information for connections that are not yet established. This is necessary in order to make them less vulnerable against *SYN flooding attacks*. As a consequence, the use of Quick-Start in <SYN,ACK> segments comes along with an increased vulnerability to SYN flooding attacks. Quick-Start is also not compatible with *SYN cookies*, which is an optional mechanism to prevent these DoS attacks.
- *Algorithms:* The actual realization of several algorithms and decisions is left open in the specification [RFC 4782]. In particular, it is not specified whether and when a sender should actually send a Quick-Start request. Another example is the question whether and how to adapt the Slow-Start Threshold after a successful Quick-Start. Different algorithms that solve these issues are proposed and evaluated in Section 6.3.3.1.

- *Violation of protocol layering*: As Quick-Start violates the layer separation between IP and TCP, interface extensions are required between the layers. For example, the Linux stack is not well prepared for sporadically added IP options. A non-trivial problem is that the TCP maximum segment size must be reduced for IP packets carrying a Quick-Start IP option in order to avoid packet fragmentation. The IP and TCP options also prevent the offloading of the checksum calculation or other protocol functions to the network card (e. g., *TCP segmentation offload*). In case of Linux, there are also no generic interfaces from the network layer to the network interface drivers from which one can determine the capacity of links, even for standard Ethernet network cards.

6 Performance evaluation

The realization of fast startup congestion control raises many issues concerning performance, fairness, and implementation costs. For end-to-end schemes, there is an inherent trade-off between a possible speedup of applications on the one hand, and the risk of packet losses and congestion on the other hand. The key question is to which extent network support can reduce or avoid these risks. Fast startup schemes also raise the question how complex their implementation is, and whether the cost is worth the effort. This chapter comprehensively studies these performance and complexity aspects. It compares both end-to-end and network-supported fast startup schemes by analytical calculations, simulation studies, laboratory experiments, as well as some tests over a real network path. The chapter is thus a novel and unique investigation of the trade-offs of fast startup congestion control. The first section presents the simulation and measurement methodology. Then, the second section verifies that results of the used simulation tool are indeed consistent with measurement results in real setups. In the third section, the fundamental behavior of different fast startup schemes is studied, and different options for several algorithms are compared, in particular for Quick-Start. The forth section analyzes the performance improvement in both synthetic and realistic workload scenarios. In the fifth section, the robustness, fairness, and risk of fast startup is explored. The chapter concludes with numerical results for the complexity and performance overhead caused by network support.

6.1 Evaluation methodology and tools

6.1.1 Simulation scenarios

6.1.1.1 Tools, topologies, and workload models

The simulation tools used in this work are based on the the *Network Simulation Cradle* (NSC), which is introduced in Section 3.3.4.3. Due to the usage of real network stack code, the resulting simulation tool-chain with the NSC is more complex than standalone simulation tools.

Figure 6.1 describes the workflow and the different steps that are needed to simulate new TCP protocol extensions. The gray boxes highlight tasks that require additional software development. The first important part is the implementation of the protocol extensions inside the TCP/IP stack. This thesis uses own Linux kernel patches for Quick-Start, Jump-Start, Initial-Start, and Mega-Start TCP extensions, which are partly described in Section 5.3.2 and also further documented in [223, 204, 205]. Some small modifications are also required in the NSC itself, in particular in order to provide handlers for additional interfaces such as new socket options and new system configuration commands. The other major part is the simulation tool that implements the network model, the host model, the application model, as well as the interfaces to the NSC as shown in Figure 3.16. Parts of the tools that are used in the following studies have been developed by Zeeh [253] and Proebster [180].

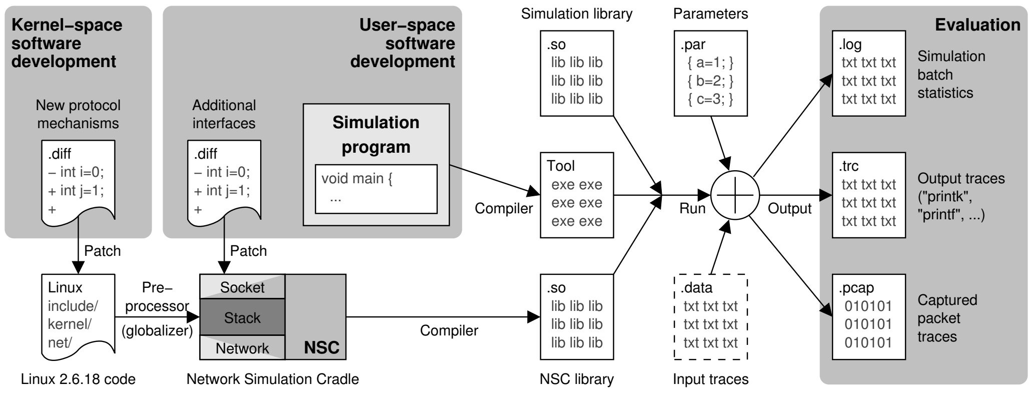


Figure 6.1: Workflow of simulations with real network stack code

In the simulation framework, the NSC-based TCP/IP stack can also be replaced by an abstract transport protocol realization that enables simulations of XCP and RCP [181]. Unlike the TCP extensions, the implementations of XCP and RCP in the simulation tool are not validated by measurements. Given the large number of open issues in the specification of these protocols, XCP and RCP experiments are only presented to round off the performance evaluation and to highlight the similarities and differences compared to the Quick-Start protocol.

It is impossible to define a set of scenarios that reflects "the typical Internet". The only realistic option is to define several standard scenarios and use them to compare different mechanisms under similar constraints. Such scenarios have been suggested for a *common TCP evaluation suite* [10]. The scenarios include several network topologies as well as a range of workload scenarios and path characteristics (RTTs, queue management schemes, etc.). Its full specification is work in progress. In the following sections, a subset of these scenarios is used. The most important one consists of the so-called *dumb-bell topology* that models a bottleneck link between two subnetworks. It is shown in Figure 6.2. Any congestion control scheme must be able to efficiently handle scare resources in such a shared bottleneck. An alternative topology consists of several subsequent routers on a path with cross-traffic (*parking lot topology*).

The dumb-bell topology is widely used as reference scenario for transport protocol evaluations and therefore also used in most of the following experiments. By default, the bottleneck

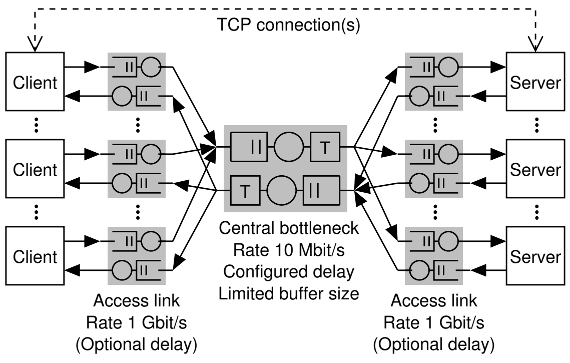


Figure 6.2: Dumb-bell simulation topology with a central bottleneck

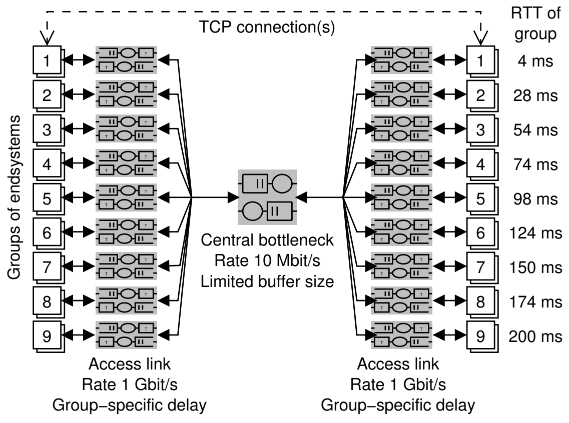


Figure 6.3: Alternative dumb-bell simulation topology with different RTTs

bandwidth is set to $r_C = 10\,\text{Mbit/s}$. This is a realistic value for the bottleneck bandwidth of long-distance connections over current access networks and WANs. Therein, the bottleneck is typically located in the access network (*last hop*). Unless stated otherwise, the central buffer has a fixed size of $B = 50$ packets and uses a drop-tail strategy. Buffers of the order of 50 packets are considered to be a realistic assumption [234]. A certain number N_{hosts} of endsystems are connected to the central bottleneck by "access links" with rate $r_A = 1\,\text{Gbit/s}$, which ensures that they cannot become the bottleneck. And there is also a configurable delay. As illustrated in Figure 6.2, homogeneous delays are enforced in the central bottleneck. Many experiments in this work consider a latency τ of the order of 100 ms or 200 ms, which are typical values for long-distance network paths. As already mentioned, TCP's flow startup is specifically challenged by such paths. An alternative setup considers nine different groups of endsystems with heterogeneous access link delays. It is depicted in Figure 6.3. This topology results in nine different RTTs between 4 ms and 200 ms, which is a realistic distribution and also recommended in the common TCP evaluation suite [10]. The MTU in all experiments in this thesis is 1500 B.

The workload is characterized by different models that reflect the characteristics of client-server applications with bi-directional data transfers realizing requests. They are described by *connection vectors* as introduced in Section 3.3.3.1. The simulations always consist of a set of connections between a client-server pair. In addition to simple models with fixed-sized requests and responses, request-response vectors are also obtained from synthetic source models. In order to model real Internet workload, traffic traces are replayed as explained in Section 3.3.3.2. The arrival of connection vectors is then modeled by an open-loop process with exponentially distributed inter-arrival times. Further specific details of the models are described in the subsequent sections. It must be emphasized that most related simulation studies in other published work only consider uni-directional traffic. Bi-directional client-server traffic reflects much more realistically the communication patterns of interactive applications and also ensures that there is traffic in the uplink direction, which reduces the risk of synchronization of flows.

6.1.1.2 Network stack configurations

The simulations are performed with the Linux kernel version 2.6.18, both without and with the new fast startup patches. This kernel version is the most recent one in the used NSC version. For the measurements, the Linux kernel 2.6.24 is used in order to have also results with a more recent stack. The network stack implementations in these two kernel versions are similar, but not identical. As a result, several subtle differences exist that also affect the results of simulations and measurements. The author tried to minimize these effects as far as possible.

Most importantly, the implementation of the TCP Slow-Start in the Linux kernel version 2.6.24 slightly differs from version 2.6.18, since the newer version also supports the Limited Slow-Start algorithm, even though it is not activated by default. The implementation of the CUBIC congestion control is also different in both kernel versions. The CUBIC implementation in kernel version 2.6.18 is known to be incorrect and has been corrected in subsequent kernel version [130, 82]. Furthermore, unlike later kernel versions, it uses an initial SST of 100 segments [82]. In order to overcome this problem, the simulation studies presented in the following use a corrected CUBIC congestion control module that has been ported back from kernel version 2.6.24. As many experiments in this work require a high-speed congestion control flavor, the CUBIC algorithms are used in all studies unless explicitly stated otherwise.

For all simulation and measurements except for the experiments in Section 6.2, the socket buffer sizes have been increased to 8 MiB, as documented in Appendix B.2. The larger buffer ensures that the flow start is never limited by the TCP flow control (cf. Section 5.2). Furthermore, the caching of connection statistics is disabled. This setting models the scenario that connections are always opened to unknown destinations. If connection caching was used, the previous SST would be reused, which would result in a better performance both of the Slow-Start and of the fast startup TCP extensions. Moreover, in the measurements the TCP enhancement "Forward RTO-Recovery" is disabled, as its realization in the Linux kernel version 2.6.24 is erroneous.

A further challenge are experiments that use persistent TCP connections: Their performance during flow restarts depends on Linux's "QuickAck" mechanism (see Section 4.2.4). After a connection setup, or after idle periods, a receiver acknowledges every segment up to an upper limit that depends, among others, on the advertised receive window. The number of "Quick-Acks" significantly affects TCP's behavior in Slow-Start. Due to receive window auto-tuning, this number may change over the lifetime of a connection. In order to make the simulation results reproducible and independent of the complex activation heuristics of the kernel, the activation threshold is manually set to a large value in this work. A side-effect of this modification is that the delayed acknowledgments are always disabled during flow start ($\eta = 1$). This means that the Slow-Start is significantly faster than a TCP stack that would use $\eta = 2$ as suggested in [RFC 2581]. In other words, the Slow-Start results presented in this work correspond to the *best possible case*. If the Linux stack did adhere to the recommendation of [RFC 2581], the fast startup schemes considered in this work would achieve a significantly larger speedup.

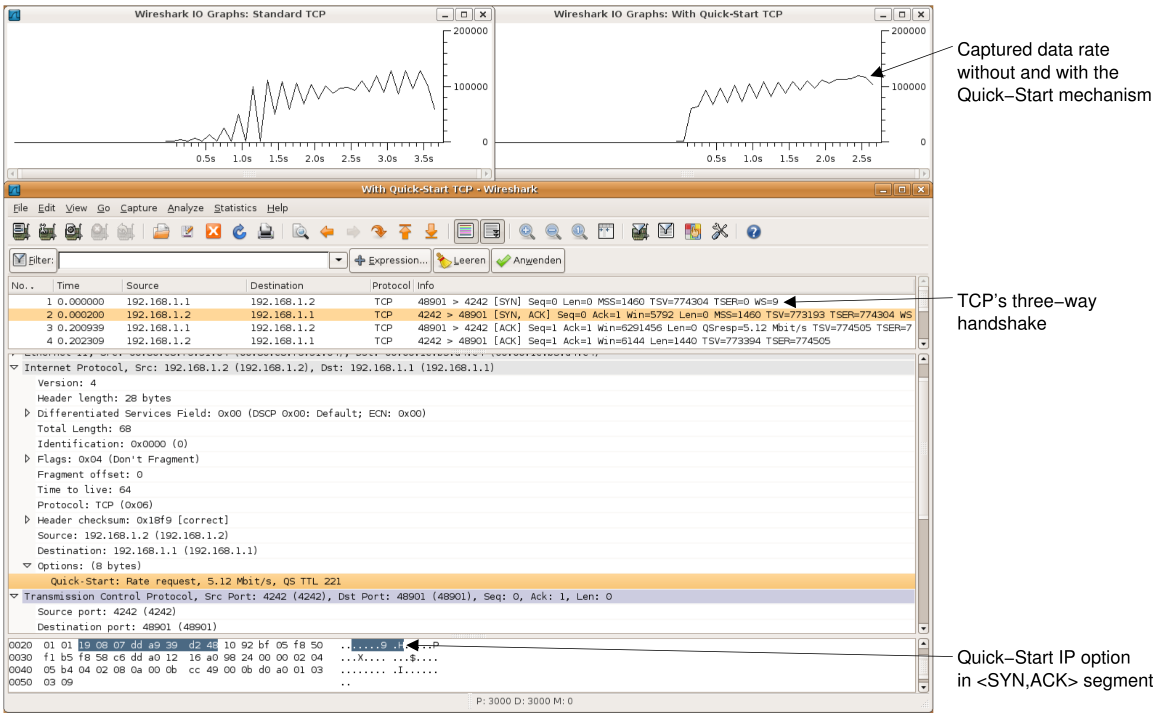
6.1.1.3 Evaluation methods

As depicted in Figure 6.1, the simulation tool provides several kinds of output in addition to statistics of the simulation environment [25]. The studies in the following sections can be subdivided into *stationary* and *transient* experiments. In the stationary case, the system is studied in its statistical equilibrium. In this steady state case, the measured values are subdivided in a transient phase and 10 batches. In addition to average values, also 95 % confidence intervals are calculated. The main steady state performance metric is the average response time. Additionally, throughput, packet loss rate, and fairness metrics are calculated. Numerical calculations outside the simulation tools are performed with the "Matlab" tool.

The transient behavior of systems with complex control loops cannot be studied by stationary experiments only. Transient experiments analyze the qualitative behavior and the response to sudden changes or events. A thorough analysis actually requires several independent replications with different initial values instead of the method of batch means. Certain effects can also be illustrated by visualizing individual traces only. The simulation framework can capture packet trace files on the network interfaces, which can then be postprocessed. One specific type of analysis used in the following are *throughput time series*. They are obtained by summing up the amount of transported data over a fixed sampling interval, which is by default 200 ms. Figure 6.4 shows an example of the analysis of a trace that contains a Quick-Start request.

6.1.2 Measurement setups

In order to verify the analytical and simulation results presented in Figure 6.4, measurements in a local network testbed have been performed. The endsystems are *Personal Computers* (PCs) with an Ubuntu 7.04 Linux operating system, either running an unmodified or a patched Linux


Wireshark IO Graphs: Standard TCP
Wireshark IO Graphs: With Quick-Start TCP
200000
100000
0
0.5s 1.0s 1.5s 2.0s 2.5s 3.0s 3.5s
Captured data rate without and with the Quick–Start mechanism
With Quick-Start TCP - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Leeren Anwenden
TCP's three–way handshake
Quick–Start IP option in <SYN,ACK> segment

```
No. .  Time       Source        Destination   Protocol Info
     1 0.000000   192.168.1.1   192.168.1.2   TCP      48901 > 4242 [SYN] Seq=0 Len=0 MSS=1460 TSV=774304 TSER=0 WS=9
     2 0.000200   192.168.1.2   192.168.1.1   TCP      4242 > 48901 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=773193 TSER=774304 WS
     3 0.200939   192.168.1.1   192.168.1.2   TCP      48901 > 4242 [ACK] Seq=1 Ack=1 Win=6291456 Len=0 QSresp=5.12 Mbit/s TSV=774505 TSER=7
     4 0.202309   192.168.1.2   192.168.1.1   TCP      4242 > 48901 [ACK] Seq=1 Ack=1 Win=6144 Len=1440 TSV=773394 TSER=774505

▽ Internet Protocol, Src: 192.168.1.2 (192.168.1.2), Dst: 192.168.1.1 (192.168.1.1)
    Version: 4
    Header length: 28 bytes
  ▷ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 68
    Identification: 0x0000 (0)
  ▷ Flags: 0x04 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
  ▷ Header checksum: 0x18f9 [correct]
    Source: 192.168.1.2 (192.168.1.2)
    Destination: 192.168.1.1 (192.168.1.1)
  ▽ Options: (8 bytes)
      Quick-Start: Rate request, 5.12 Mbit/s, QS TTL 221
▽ Transmission Control Protocol, Src Port: 4242 (4242), Dst Port: 48901 (48901), Seq: 0, Ack: 1, Len: 0
    Source port: 4242 (4242)
    Destination port: 48901 (48901)

0020  01 01 19 08 07 dd a9 39  d2 48 10 92 bf 05 f8 50   ........9 .H...P
0030  f1 b5 f8 58 c6 dd a0 12  16 a0 98 24 00 00 02 04   ...X.... ...$....
0040  05 b4 04 02 08 0a 00 0b  cc 49 00 0b d0 a0 01 03   ........ .I......
0050  03 09                                              ..

P: 3000 D: 3000 M: 0
```

Figure 6.4: Quick-Start dispatcher in the "Wireshark" tool that displays traces

kernel version 2.6.24. The basic testbed setup is sketched in Figure 6.5 and also documented in Appendix B.1. The client and server applications run on computers that are interconnected by Ethernet segments, which are by default manually throttled to a line speed of 10 Mbit/s in order to emulate a realistic Internet long-distance path capacity. The Ethernet flow control is disabled in order to avoid interactions with the congestion control. *TCP segmentation offload* is also disabled (cf. Section 5.3.4). The Linux network emulation [90] is used in order to enforce constant minimum packet delays. The network stack configuration is, as far as possible, identical to the simulation settings explained in Section 6.1.1.2.

When real equipment is used, it is difficult to exactly control the amount of available buffer space on a path, since buffers do not only exist inside the IP network stack, but also in the network interface card, but can hardly be controlled there. This is why the impact of limited buffer space is mostly evaluated by simulation only.

Fast startup congestion control is particularly interesting for interactive applications that frequently exchange certain amounts of data and thus often suffer from the Slow-Start. The exper-

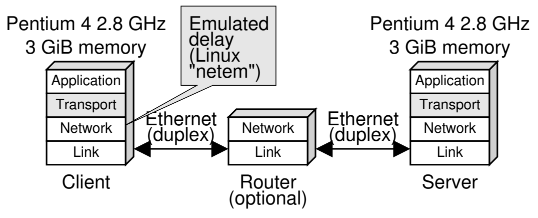


Figure 6.5: Experimental lab setup with several computers

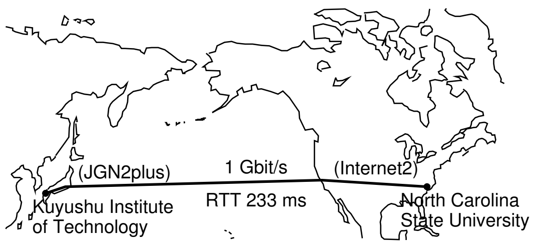

(JGN2plus)
1 Gbit/s
(Internet2)
Kuyushu Institute of Technology
RTT 233 ms
North Carolina State University

Figure 6.6: Illustration of the experimental path used in some experiments

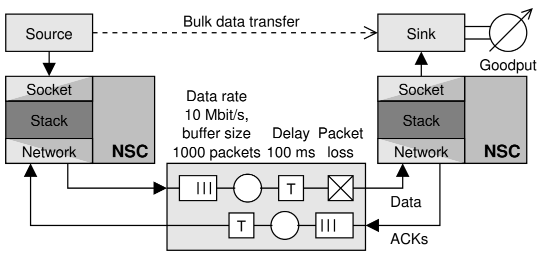


Figure 6.7: Setup of the validation scenario that studies the response function

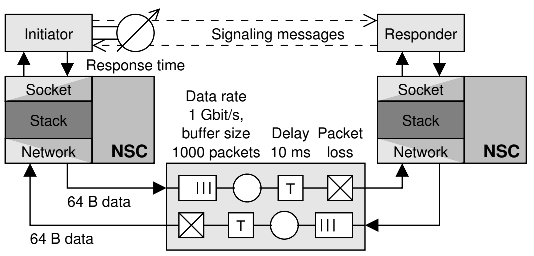


Figure 6.8: Setup of the validation scenario for Head-of-Line blocking

iments use two different types of interactive applications: First, tests are performed with simple client and server C programs that use straightforward socket calls. The server is also able to explicitly activate a fast startup by the expanded sockets interfaces, if required. The second class of tests consists of a modified “lighttpd” Web server (version 1.4.18), which can optionally explicitly activate a fast startup [223]. The workload is then generated by the SURGE Web traffic generator (version 1.00a), which is introduced in Section 3.3.3.1, by the “httperf” tool, or directly by the “Firefox” Web browser. Long-lived flows that emulate long file transfers are generated by the “iperf” tool. Finally, stress tests with high workloads are realized by using an Agilent network analyzer for workload generation and high-precision tracing.

6.1.3 Experiments in a test network

A laboratory experiment can hardly represent a complete Internet path. This is why some experiments have also been performed over a real long-distance network path shown in Figure 6.6. The endsystems as well as the path form part of an experimental network testbed that is used for transport protocol performance tests. The path can also be traversed by packets that carry new IP options. The bandwidth of the path is about 1 Gbit/s and the RTT is 233 ms. The main advantage of using this infrastructure is that it allows experiments under real constraints, e. g., concerning router buffer sizes, without the need to run a network emulation. However, because this infrastructure could only temporarily be accessed, only few experiments could be realized in this environment.

6.2 Simulation tool validation

6.2.1 Scenario selection

As explained in Section 3.3.4.1, any simulation study must be validated. The following subsections verify that the used simulation tool indeed provides results that are close to measurements in real setups. The two validation scenarios shown in Figures 6.7 and 6.8 study the simulation accuracy of the throughput and transport delay, respectively. Both metrics are compared to real-world measurements and analytical approximations. These experiments complement other NSC validation tests of Jansen [103, 104], who has not systematically compared the simulation results to models and who has also published only few results on the simulation accuracy compared to measurements.

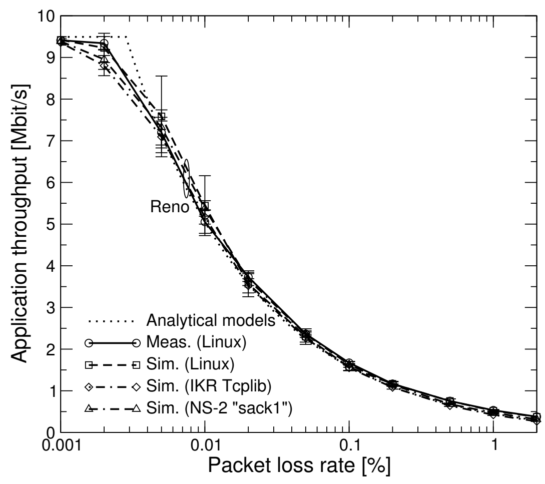


Figure 6.9: Simulated Reno response functions vs. models and measurements

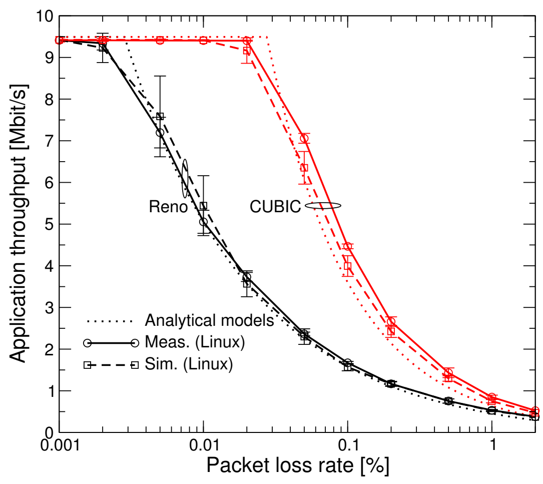


Figure 6.10: Simulated CUBIC response functions vs. models and measurements

6.2.2 Validation scenario: Response function

6.2.2.1 Motivation

The first validation scenario studies the *response function* $G(p)$, which quantifies the throughput G of a single connection as a function of the packet loss probability p for a given RTT. This function describes the fundamental behavior of a congestion control algorithm. A closed-form expression for the Reno algorithm is provided in Equation (4.7). Corresponding terms for the CUBIC and Compound congestion control are given in Equation (4.9) and Equation (4.10), respectively. As shown in Figure 6.7, the simulation setup consists of one TCP connection with a greedy source running over an emulated 10 Mbit/s Ethernet Link. In the measurement, the corresponding scenario is set up with two computers as illustrated in Figure 6.5.

6.2.2.2 Comparison of simulation and measurement results

Figure 6.9, Figure 6.10, and Figure 6.11 study how packet loss affects the TCP throughput for an RTT of $\tau = 200\,\mathrm{ms}$. The simulation results obtained from the NSC-based tool are compared to measurements as well as to the theoretical models. Reno, CUBIC, and Compound are used as congestion control algorithms. In case of Reno, two other simulation tools are considered as further references: NS-2 version 2.31 with the “sack1” TCP model and the IKR Tcplib version 1.4.3. These two simulation tools provide by default a Reno congestion control only.

The graphs in Figures 6.9, 6.10, and 6.11 illustrate three different aspects: First, the response functions all have a characteristical shape. For a low packet loss rate $p \ll 1\,\%$, the throughput is close to the link capacity. Due to the overhead of the Ethernet header, inter-frame gaps, and the IP and TCP header the maximum value is smaller than the Ethernet link capacity of 10 Mbit/s. In the simulation, the IP data rate is therefore set to $r = 9.76\,\mathrm{Mbit/s} \approx 10\,\mathrm{Mbit/s}$. This value is the effective capacity of a 10 Mbit/s Ethernet link for full-sized packets with $MTU = 1500\,\mathrm{B}$. The slight reduction is caused by the Ethernet header and inter-frame gaps. It is considered in all simulation setups in this thesis.

When the packet loss rate p gets larger, the congestion control more and more limits the throughput. The second observation is that the high-speed congestion control schemes CUBIC and

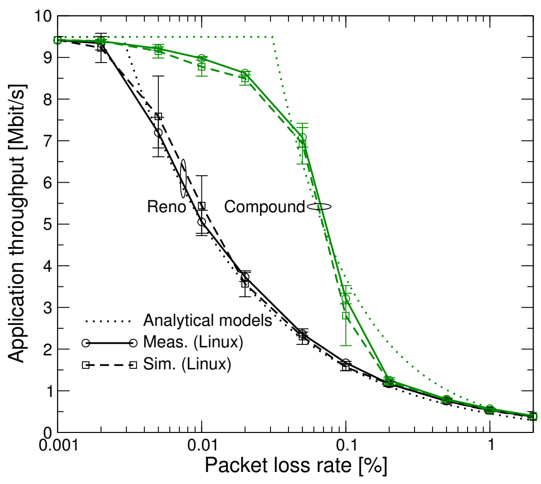


Figure 6.11: Simulated Compound response functions vs. models and measurements

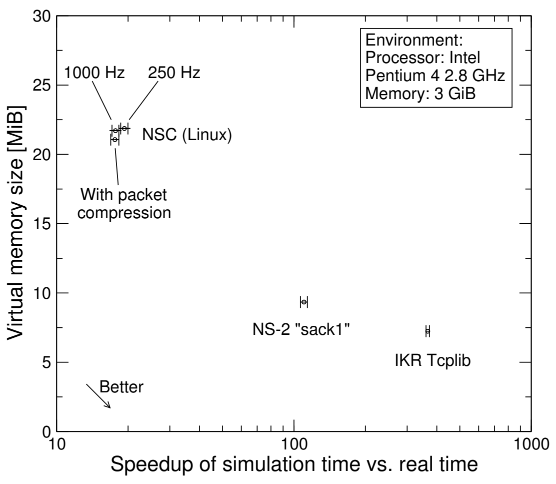


Figure 6.12: Runtime performance comparison of the different simulation tools

Compound result in a significantly larger throughput for moderate packet loss rates of the order of 0.1 %. If the packet loss rate is larger, they fall back to a Reno-like behavior. Both effects exactly correspond to the design objectives of high-speed congestion control algorithms. For packet loss rates of the order of 1 %, the average CWND is kept at a rather small value and a TCP connection is unable to completely utilize a path with a larger BDP.

Third, the simulation results and the measurements reveal a rather close match. All Reno simulations are also almost identical to the model of Equation (4.7). The models for CUBIC and Compound are also good approximations in particular for moderate packet loss rates. These results prove that an NSC-based simulation tool indeed delivers a throughput that differs by at most few percent from the value that one would measure in a corresponding real experiment.

6.2.2.3 *Runtime performance and scalability limits*

The execution of real network stack comes at some cost, which is quantified in Figure 6.12. This figure depicts two runtime performance metrics. It corresponds to simulations of the scenario shown in Figure 6.7 with Reno congestion control and a packet loss rate of $p = 0.1\,\%$. The diagram reveals two properties of NSC-based simulators:

- *Simulation speed*: In small and mid-sized scenarios a NSC-based simulation runs significantly faster than real time. As shown in Figure 6.12, a measurement with equivalent number of samples would have required at least 20 times the duration of the simulation. Still, the NSC-based simulation is about one order of magnitude slower than a simulator that use a simplified model of TCP only. This is a well-known drawback of the NSC, and a slowdown of the same order of magnitude is also reported by Jansen [103]. The simulation speed also decreases approximately linearly with the number of instantiated network stacks, since the timer interrupt must be triggered in each stack and causes processing overhead.
- *Memory consumption*: Simulations with the NSC require a significant amount of memory. The largest part is consumed by the shared library that contains the kernel code. As this library must be loaded only once in homogeneous simulation scenarios, it results in a constant minimum memory requirement. A certain additional amount of memory is

required per stack, mainly for providing the buffer space. Finally, memory is required for storing the outstanding packets on the path. In the latter two cases, the exact amount of memory for depends very much on the simulation scenario.

The memory requirements and the usage of interrupt timers impose an upper *scalability limit of the order of thousand stacks* per simulation tool. There are also techniques how to improve the scalability [253, 200]. Most importantly, the timer interrupt frequency could be reduced. For instance, a Linux kernel uses by default a frequency of $HZ = 1000\,\mathrm{Hz}$, but it can also be configured to run with 250 Hz. Furthermore, the memory consumption can be reduced by using pseudo data as payload of packets [253]. In this case the simulator does not have to allocate a whole MTU of memory for each packet. As shown in Figure 6.12, both methods indeed improve the performance, but their impact is small. The usage of NSC-based simulation tools in very large simulation scenarios with more than thousand stacks would require further optimizations.

6.2.3 Validation scenario: Head-of-line blocking

6.2.3.1 Motivation

The throughput measurements in the previous section do not provide insight into the timing accuracy of the simulation tool. Therefore, a second validation scenario studies how precisely an NSC-based simulation tool models latencies. The considered use case is a transaction-based signaling application that exchanges small signaling messages between two signaling entities over a path that is subject to symmetric packet losses. This kind of workload is typical for control plane traffic, e. g., for firewall control [116]. The corresponding simulation setup is depicted in Figure 6.8. The initiator periodically sends request messages of length $s = 64\,\mathrm{B}$ with a negative exponentially distributed IAT with mean value $\delta = 10\,\mathrm{ms}$. The messages are sent over a single TCP connection to the responder, which echos back the messages. The initiator then measures the response time of each request-response pair. In this experiment it is important that both endsystems use the `TCP_NODELAY` socket option. Due to the usage of bidirectional data transfers, this scenario cannot easily be realized in simulation tools that only model unidirectional communication.

TCP provides an ordered reliable transport service, even though reliability and ordered delivery are actually orthogonal issues. As a consequence, *Head-of-Line Blocking* (HOL) occurs when packets get lost, as subsequent messages have to wait for the successful retransmission in the receiver queue and are thus delayed. The impact of HOL on the response time of transaction-based signaling applications has been analyzed analytically [197, 116] and can thus be used in order to validate the simulation tool. The analytical model is briefly summarized in Section A.2.

6.2.3.2 Comparison of simulation and measurement results

Figure 6.13 shows the mean response time as a function of the packet loss rate for the Reno and CUBIC congestion control. Again, the simulation results and the measurements are very similar, except for a constant offset of about 2 ms. This difference can be attributed to the network emulation which adds an error of about 1 ms per direction. As long as the packet loss rate is small, the results are also close to the reference value given by Equation (A.7) with the processing overhead ε assumed to be 0.5 ms. However, for $p > 1\,\%$, both the simulated and measured mean response time is larger than predicted by the model. This can be explained by

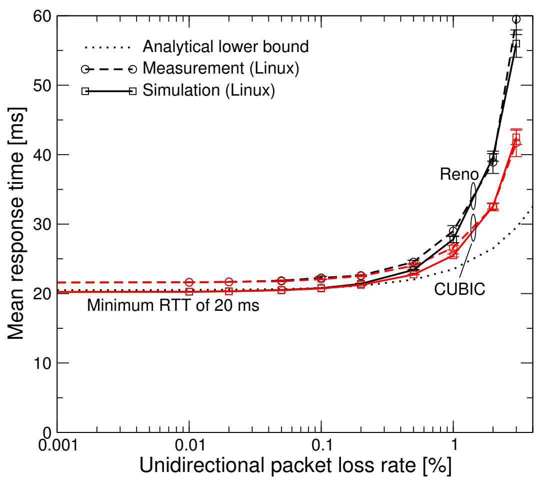


Figure 6.13: Comparison of the impact of HOL in simulation and measurements

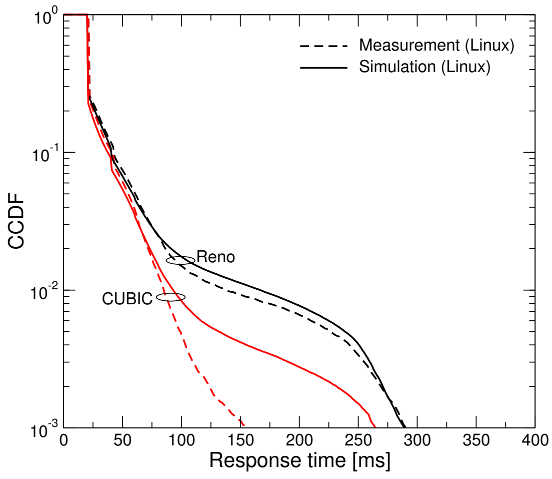


Figure 6.14: Response time distribution resulting from HOL

the TCP congestion control that limits the throughput when losses occur frequently. As to be expected, CUBIC outperforms Reno in this parameter range.

Despite the close match of the mean values of the simulation and measurement results, there are also some differences. They can be observed in the CCDF for $p = 1\,\%$ that is depicted in Figure 6.14. The CCDF reveals a non-negligible probability for high delays. In case of Reno, simulation and measurement results are consistent. But there are some differences in the distribution tail for the CUBIC congestion control. The delay obtained from the simulations and measurements differs in about one percent of the samples. This slight discrepancy might be caused by small implementation differences in the network stacks used for the simulations and for the measurements, which can have an impact on the timing of certain packets (e. g., error recovery algorithms). But as this inaccuracy only affects a very small number of samples, it has hardly any impact on experiments that consider mainly mean values.

6.2.4 Summary of the validation experiments

All in all, the validation experiments show a rather close match of the simulation results and measurements in a comparable setup. Compared to simulation tools with simplified TCP models, the main shortcomings of NSC-based simulations with real kernel code are longer simulation durations and a significantly larger memory consumption. These results confirm validation experiments of Jansen who also found differences of the order of few percent only [103, 104]. Yet, the observed simulation and measurement results can never be expected to be identical because of a number of reasons:

- The application interface differs between a simulation tool and a real operating system. It is difficult to precisely model delays caused by the interaction of the network stack and application, even though there have been efforts [253]. *Blocking socket calls* are difficult to model. In a message-oriented simulator, they have to be replaced by a message interface that may behave differently. Also, different write patterns can affect the memory allocation and buffer handling. For instance, the application write patterns affect the usage of the “PUSH” flag. These effects could only be simulated by a workload model that exactly reflects the read and write operations of an application.

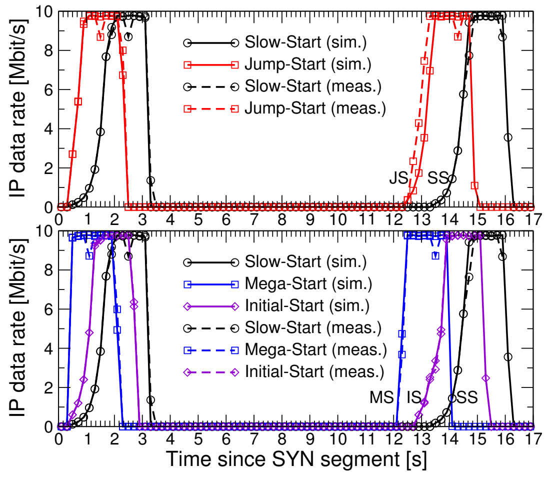


Figure 6.15: Fast startup and fast restart of different end-to-end mechanisms

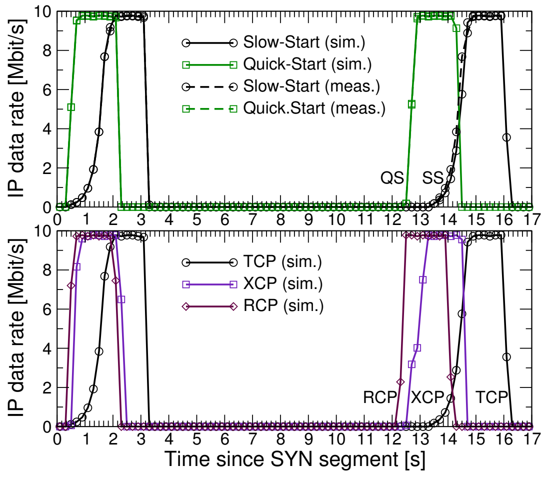


Figure 6.16: Fast startup and fast restart of different network-supported mechanisms

- The simulation model does not model the network interface card and its buffer. Therefore, it also does not model local backpressure mechanisms if this buffer is full.
- There are many sources of other delays, such as competing processes and the scheduler of the operating system. These delays cannot easily be modeled in a simulation.
- As explained in Section 6.1.1.2, there are small differences between the kernel versions used in the simulation (version 2.6.18) and the one used in measurements (version 2.6.24). This is an inherent problem because real TCP/IP stacks are always a “moving target”.

One has to be aware of these effects when using simulations with real network stacks, even though their impact can be neglected in many cases.

6.3 Study of functional design aspects of fast startups

6.3.1 Comparison of the startup behavior

In the following, the different end-to-end and network-supported fast startup congestion control schemes introduced in Sections 4.4, 4.5, and 4.6 are studied. The fundamental properties of Initial-Start, Jump-Start, Mega-Start, Quick-Start, XCP, and RCP can be examined in a simple and deterministic scenario: It consists of one client and one server ($N_{\text{hosts}} = 1$), a single bottleneck, and no other traffic. In order to validate the implementations in the simulation tool, this scenario has been setup up both in simulations and in the lab testbed explained in Figure 6.5. Figures 6.15 and 6.16 show the throughput at IP layer for a simple client-server transaction with the connection vector $\mathbf{V} = [(100, 2\cdot 10^6, 10), (100, 2\cdot 10^6, 0)]$, i. e., two requests of size 100 B and two responses of 2 MB with an idle time of 10 s. This corresponds to the typical *on-off communication pattern* of interactive applications. In the simulation, the IP data rate is set to $r = 9.76\,\text{Mbit/s} \approx 10\,\text{Mbit/s}$, which is the effective capacity of a 10 Mbit/s Ethernet link that is used in the measurements. The presented traces are measured in downlink direction at the client. They have been obtained by summing up the size of the received IP packets within slots of a duration that is equal to the minimum RTT $\tau = 200\,\text{ms}$. For the comparison of simulations and measurements, the buffer size is set here to $B = 1000$ packets, which is the default IP layer buffer size in the Linux kernel. The impact of smaller buffer sizes is studied in the next sections.

Figure 6.15 reveals that the Slow-Start algorithm needs about 2 s until the path can be fully utilized in such a setup, both after connection setup and after a long idle time. In the latter case, the Congestion Window Validation has reduced the window to the *restart window*. As to be expected, all fast startup mechanisms are much better. The upper part of the diagram presents the results for Jump-Start, which plays out up to $K_{\text{data}} = 64\,\text{KiB}$ during the first RTT. One can also observe that the implementation of Jump-Start correctly detects the long idle time and activates a fast restart.

In the lower part of Figure 6.15, the performance of the Mega-Start scheme is shown under the assumption that the sender approximately knows the available bandwidth of $r = 10\,\text{Mbit/s}$ and uses such an initial rate. This graph represents the theoretical optimum in this scenario. Furthermore, just increasing the initial window to $w_{\text{IS}} = 10$ would also work in this scenario (Initial-Start), even though the speedup is smaller. In this specific path with significant buffer space, it would even be possible to use larger initial values without risking packet loss. If the buffer had been smaller, overflow would have occured, and the resulting completion time would be similar to Slow-Start or even worse.

Figure 6.16 shows comparable results for different network-supported schemes, with the Slow-Start as a reference: On a 10 Mbit/s link, the maximum useful Quick-Start request rate is $q_{\text{req}} = 5.12\,\text{Mbit/s}$ [RFC 4782]. In this example, the server piggybacks such a request on the <SYN,ACK> segment (*early activation*) and reissues it after the idle time. As the path is empty, both requests are approved, and the data transfer starts with this rate. The upper part of Figure 6.16 shows the resulting trace. As one can see, Quick-Start ramps up to full link speed in about 300 ms. In this diagram, the sender does not adapt the Slow-Start Threshold after the Quick-Start validation phase, i. e., it continues in Slow-Start mode. This specific design choice is investigated more in detail in Section 6.3.3.1.

Figure 6.16 also illustrates corresponding simulation results for XCP and RCP. The performance of RCP is quite similar to Quick-Start, whereas the speedup of XCP is slightly smaller. Concerning RCP, some assumptions were necessary in order to realize such a bi-directional communication with idle times. The published descriptions of RCP [58, 57] do not precisely specify how the protocol could handle idle times. The RCP model in this thesis uses the last valid feedback value for flow restart and thereby differs from the NS-2 model that sends unnecessary probe traffic instead [181]. Obviously, this algorithm can be very aggressive after idle times. An alternative would be to restart flows with a small initial rate, as it is realized in case of XCP.

As to be expected, the comparison of Figures 6.15 and 6.16 shows that the performance of all fast startup schemes is similar in this specific setup. Yet, there are some subtle differences: First, the different schemes depend to a different degree on the measured RTT. In the given setup, the last measured value after a long idle time is larger than the actual RTT, which has been inflated by queueing during the previous data transfers. Rate-based approaches (Quick-Start, Mega-Start, etc.) are quite insensitive to RTT measurement errors. If the estimated RTT is too large, acknowledgments arrive during the rate pacing phase, which must be stopped early. Still, segments are played out with the expected rate. In contrast, Jump-Start is sensitive with respect to inflated RTTs and reacts by a reduction of the sending rate. This effect can be observed in Figure 6.15.

Second, Quick-Start (as well as XCP) requires signaling before it can ramp up the rate. This signaling results in an additional delay after a long idle time. When application data becomes available, the sender can only send as much data as permitted by the *restart window*. For

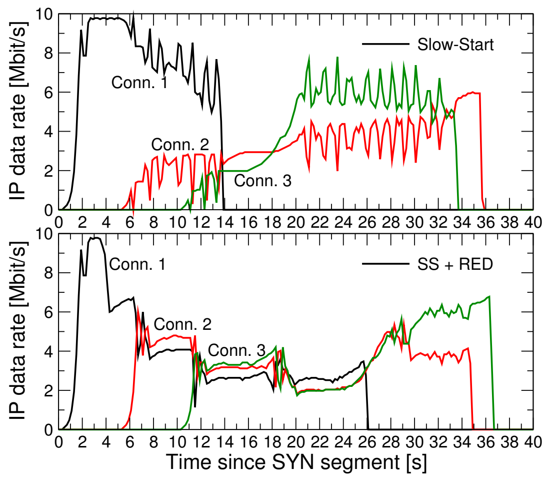


Figure 6.17: Behavior of competing flows using Slow-Start without and with RED

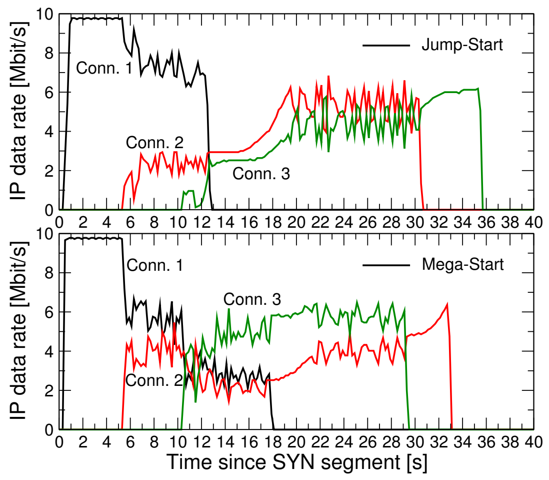


Figure 6.18: Behavior of competing flow using Jump-Start or Mega-Start

increasing the sending rate beyond this value, the source must wait one RTT and receive the feedback. Compared to this, end-to-end fast startup scheme can start to send immediately. This additional signaling delay can be observed in Figures 6.15 and 6.16 by comparing Quick-Start and Mega-Start: After the idle time, Quick-Start uses a high sending rate about 200 ms later.

For all schemes except RCP and XCP, Figures 6.15 and 6.16 also include traces that have been measured in a testbed with real computers. The comparison with simulation results shows only small differences. This again backs up the validity of the simulation results.

6.3.2 Differences in convergence and bandwidth sharing

One of the most important characteristics of a congestion control scheme is the handling of *competing flows*. A bottleneck link is seldomly used by a single flow only. The *convergence behavior* and fairness characteristics of the different fast startup schemes can be compared in a similar simulation scenario with $N_{\text{hosts}} = 3$ clients and servers ($r = 9.76\,\text{Mbit/s}$, $\tau = 200\,\text{ms}$). The workload for each client/server pair is described by a connection vector $\mathbf{V} = [(100, 12000000, 0)]$ and a start offset of 5 s between the different connections. This specific scenario is selected since it results in different flow arrival and departure patterns. Other fairness experiments with statistical workload patterns are presented later in Section 6.5. In order to allow the congestion control to converge, the drop tail buffer in the central bottleneck now has a size of $B = 50$ packets. Figures 6.17, 6.18, 6.19, and 6.20 show throughput traces of the three connections if all use the same flow startup principle.

The microscopic behavior of congestion control algorithms in such a transient scenario is very sensitive to the path characteristics and the queue management schemes. Even small changes can result in a completely different transient behavior. The presented traces can therefore only illustrate one possible outcome of the experiment. Nonetheless, several fundamental differences of the convergence properties can be observed: The graphs in the upper part of Figure 6.17 show that the CUBIC congestion control, as well as most other TCP congestion control flavors, does not necessarily result in equal rates for each flows, even if the RTT is equal. In the given example, both flows that start later are unable to reach an equal share of the link capacity, which

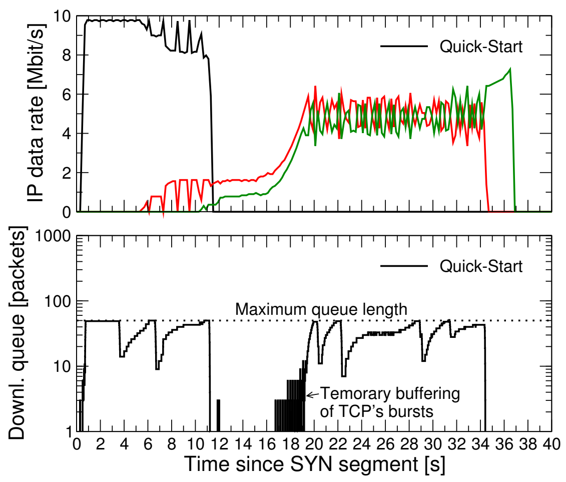


Figure 6.19: Behavior of competing flows using Quick-Start with limited buffer

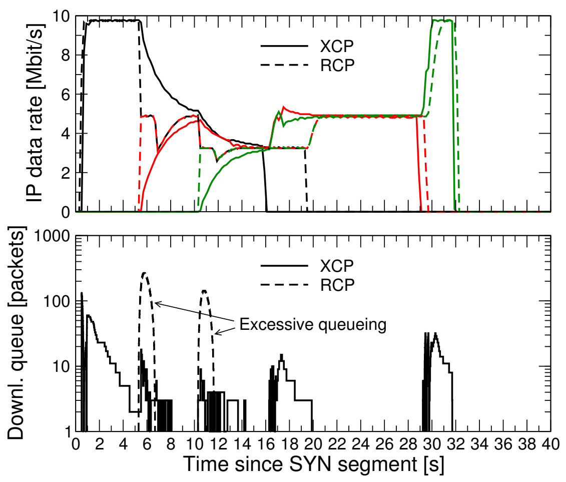


Figure 6.20: Behavior of competing flows using XCP or RCP with unlimited buffer

is mostly occupied by the first flow. This lack of convergence to equal rates is typical for TCP and bottlenecks with drop tail buffers.

As already explained in Section 4.2.2.2, the fairness can be improved by *Active Queue Management* (AQM) mechanisms such as *Random Early Detection* (RED) [RFC 2309]. In the lower chart of Figure 6.17, the central bottleneck uses an RED strategy instead of drop tail. The advantages are evident: The flows have a roughly equal throughput most of the time, and there are much less oscillations. Another effect of AQM is that the specific congestion control algorithms used by the endsystems have less impact on the total performance. But since AQM is still not predominantly used in the Internet, the following experiments use drop tail buffers.

The experiment shown in Figure 6.18, as well as other studies published in [205], reveal that a fast startup with Jump-Start does not necessarily improve the convergence to equal bandwidth sharing. Given that Jump-Start reduces its CWND if packet losses occur, it depends on the loss pattern whether it is able to take bandwidth from long-lived flows. A fast startup must send almost with full path capacity (Mega-Start) in order to improve the performance of the new flows at cost of the already established connections. As explained in Section 4.3.1, it is an open and controversial question whether such an aggressive behavior of short flows is indeed fair [172]. This thesis argues similar like Briscoe [35] that it is *reasonable to accept some short-term performance degradation of long-lived flows* if this speeds up short and mid-sized transfers. With high-speed congestion control, long-lived flows are anyway able to return to their previous rate within few seconds, so that the influence on their overall performance is very small.

Figure 6.19 reveals that Quick-Start does not improve the convergence behavior if the routers use an admission control strategy that prevents oversubscription of bandwidth. Due to lack of available bandwidth, the Quick-Start requests are denied at the bottleneck, and therefore any new connection falls back to Slow-Start. The other chart in Figure 6.19 depicts the evolution of the queue length in downlink direction. As to be expected, the long-lived flows cause the queue to be filled most of the time. If the link is not completely utilized, the bursty arrival of packets cause short-term fluctuations of the queue length. This is an inherent effect of a window-based transport protocol.

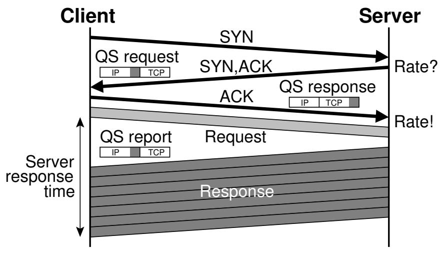


Figure 6.21: Client-server communication with early QS activation

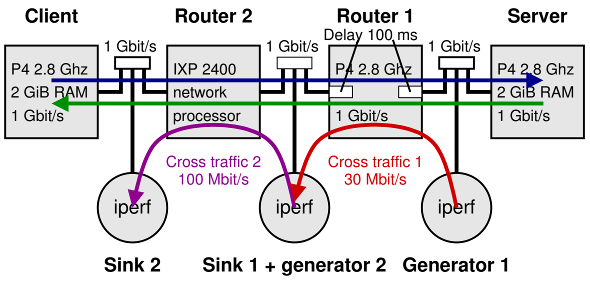


Figure 6.22: Quick-Start testbed with different router implementations

The behavior of XCP and RCP is completely different in such a setup: As illustrated in Figure 6.20, both network-controlled schemes indeed converge to equal bandwidth sharing, but at different time scales: XCP assigns bandwidths to new flows rather slowly. As a consequence, the convergence time is of the order of 5 s. Being a window-based protocol, XCP also requires some temporary buffering. RCP is much faster and achieves equal throughputs within one RTT. This advantage of RCP is well-known [57]. However, RCP requires a very large amount of buffering when new flows arrive. The required buffer space can be of the order of several hundred packets or even larger, which is often not available in current network components. This effect is also known from other studies [102]. The simulation results in Figure 6.20 assume that the required buffer space is indeed available ($B = \infty$). If B had been small, packet loss would have occured, and both schemes would have to use TCP-like mechanisms that are not specified so far.

6.3.3 Design and evaluation of new algorithms for Quick-Start

6.3.3.1 End-system functions

The following sections study some algorithmic aspects that are specific to the Quick-Start protocol. A unique characteristic of Quick-Start is that the endsystems have to request for a certain data rate q_{req} as illustrated in Figure 6.21. This raises the question of *when to request for what rate*. A very important issue is to avoid needless requests. As a solution, an intelligent activation strategy is proposed in Section 4.5.3.5. Furthermore, the requested rate can either be larger or smaller than the available bandwidth on the path. In both cases there are design choices that affect the performance: If the requested bandwidth is larger than the available bandwidth on a path with several routers as shown in Figure 6.22, the approval control algorithms have a significant influence, which is studied in the next subsections.

If the approved rate is smaller than the available bandwidth, Quick-Start may not be able to immediately utilize the path. In this context, an important question is whether to adapt the Slow-Start Threshold (SST) after a successful Quick-Start probing phase, or not. This question is left open by the Quick-Start specification [RFC 4782]. Several alternative strategies are possible:

- *Strategy a* (or QS-a): The SST is not adapted after a successful Quick-Start request, i. e., it remains at its default initial value. In case of Linux, the initial value is very large and the sender thus continues with the Slow-Start algorithms after the rate pacing.
- *Strategy b* (or QS-b): The SST is set to the Quick-Start window or some multiple of it. As a consequence, the data transfer continues in Congestion Avoidance. The rational behind

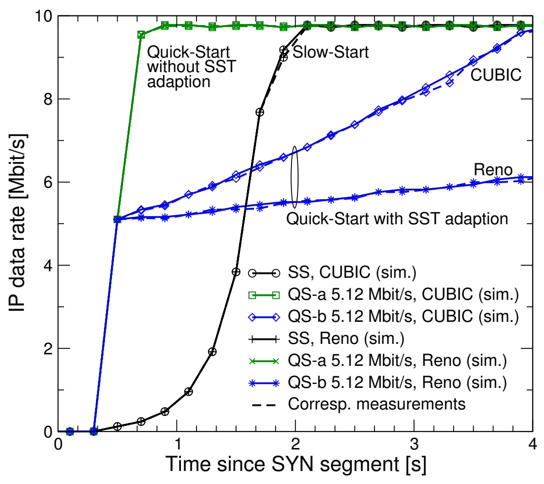


Figure 6.23: Quick-Start with different SST adaptation strategies on a 10 Mbit/s path

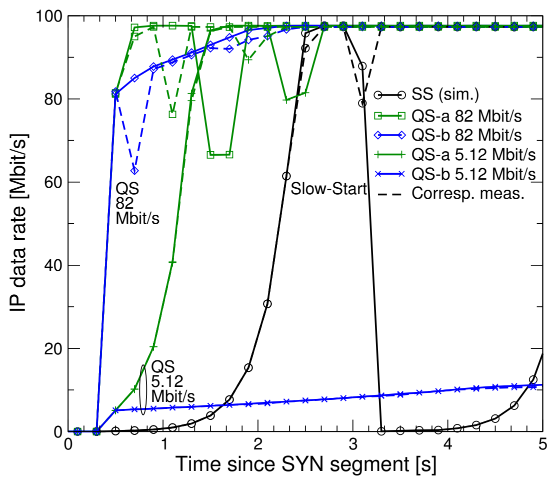


Figure 6.24: Quick-Start with different SST adaptation strategies on a 100 Mbit/s path

this is to use the information about the path characteristic obtained from the Quick-Start mechanisms. This algorithm is recommended in [190].

- *Strategy c* (or QS-c): The SST is adapted to the Quick-Start window only if the requested rate is explicitly set by the application and if this rate is also approved. This strategy combines aspects of the other two ones and has not been proposed elsewhere.

All these strategies have advantages and drawbacks. In strategy QS-b and potentially also in strategy QS-c, the sender continues in Congestion Avoidance. This variant is less aggressive and does not risk *Slow-Start overshooting*. However, a reduction of the SST is also disadvantageous in many situations. If the approved rate q is smaller than the path capacity, the sender enters the Congestion Avoidance phase before the sending rate reaches the link capacity. Hence, it takes a certain additional time until the available bandwidth is indeed utilized. If the Reno algorithms are used instead of a high-speed congestion control flavor, this time can be extremely long.

This effect is confirmed in Figure 6.23, which prints the first seconds of a long download from a server. The parameters are: $\mathbf{V} = [(100, 100 \cdot 10^6, 0)]$, $r = 10\,\text{Mbit/s}$, $\tau = 200\,\text{ms}$, $B = 1000$. Even with CUBIC it is possible that the Slow-Start is faster than Quick-Start in combination with strategy QS-b. In contrast, the link can almost instantaneously be utilized if the SST is not adapted (QS-a). The drawback of the SST reduction is even more apparent if the path capacity is larger, as shown in Figure 6.24. With QS-a, the connection can almost immediately utilize the link with the capacity 100 Mbit/s and an RTT of $\tau = 200\,\text{ms}$, independent of the approved rate. But with QS-b, even CUBIC increases the data rate very slowly beyond an approved rate of 5.12 Mbit/s. This shows that strategy QS-b hardly makes sense if $q \ll r$. One exception could be the case that an application explicitly states that a sending rate of q is indeed sufficient.

In summary, not adapting the SST after a successful Quick-Start results in faster data transfers in many situations. Therefore, the strategy QS-a is the default choice in the following experiments, unless stated otherwise. Strategy QS-b, i. e., an unconditional adaptation of the SST, cannot be recommended. A possible alternative could be strategy QS-c, in particular if applications would be able to specify whether the requested value is the maximum useful data rate. A corresponding API extension is suggested in Section 5.1.1.3. As long as applications cannot precisely specify their requirements, QS-a is a more generally applicable strategy.

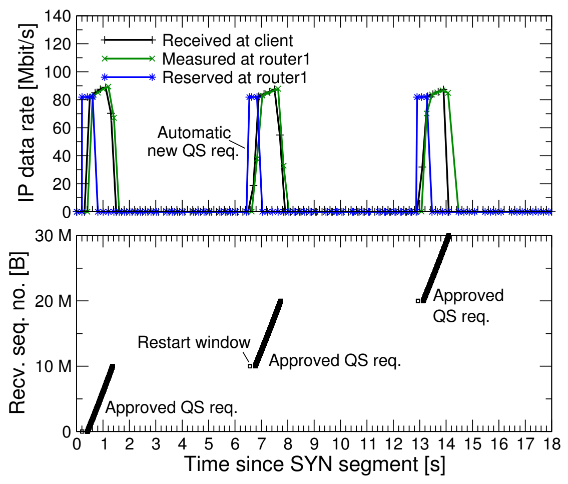


Figure 6.25: Quick-Start on an emulated path with several routers without cross-traffic

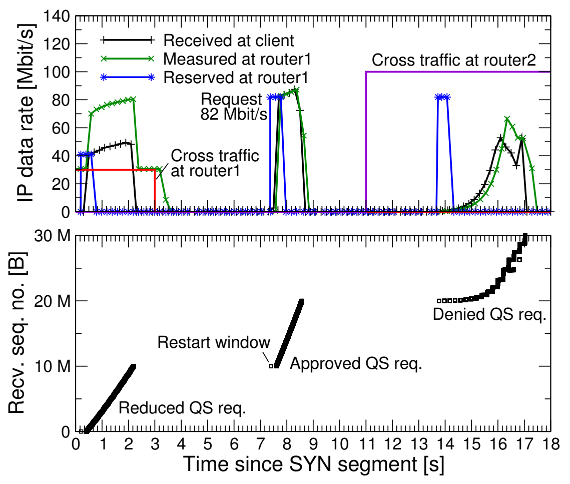


Figure 6.26: Quick-Start on an emulated path with several routers with cross-traffic

Unlike in all previous experiments, Figure 6.24 also reveals some differences between the simulation and measurement results after the flow startup phase. In the simulation, the unmodified network stack with Slow-Start suffers from a retransmission timeout after about 3 s, which does not occur in the measurements. The trace originating from measurements and simulations also differ if Quick-Start is enabled. This effect can be explained by a subtle difference between the simulated network stack and the laboratory setup shown in Figure 6.5: As already mentioned in Section 6.2.4, the Linux network stack in the endsystem has a local *backpressure mechanism* from the link layer to the transport layer. If the buffer in the local interface card is full and TCP tries to send further packets, an error code is sent back. The Linux TCP implementation reacts in this case as if this packet had been lost and triggers the congestion control. However, in the simulation the network interface buffer is not modeled inside the kernel. As a consequence, the TCP network stack does not receive these backpressure signals. The simulation tool can therefore not accurately study scenarios where the bottleneck is in the first hop. This difference is less important in dumb-bell topologies in which the first hop can never become a bottleneck.

6.3.3.2 *Fundamental behavior of the target algorithm approval control*

A request for a too large rate triggers the approval control in one or more routers on the path. Figure 6.22 illustrates a *parking lot topology* with several routers. In this case, these routers even use different implementations. In order to demonstrate the operation of the approval control with the target algorithm (cf. Section 4.5.2.4), the client downloads three times a data volume of 10 MB with 5 s idle time over a persistent TCP connection. In all entities, the Quick-Start approval threshold is set to $\theta \cdot c = 100$ Mbit/s. The requested Quick-Start rate is $q_{\text{req}} = 82$ Mbit/s, and the endsystems use QS-b for illustration purposes. In order to challenge the approval control, further inelastic traffic sources have been set up. If they are enabled, a cross traffic of 30 Mbit/s interferes with the first download, and during the third download there is a background load of 100 Mbit/s.

Figure 6.25 and Figure 6.26 show the resulting communication as observed both by the client and router 1. In router 1, both the currently carried traffic and the Quick-Start request history has been extracted from kernel logging messages. If there is no cross-traffic, all requests should

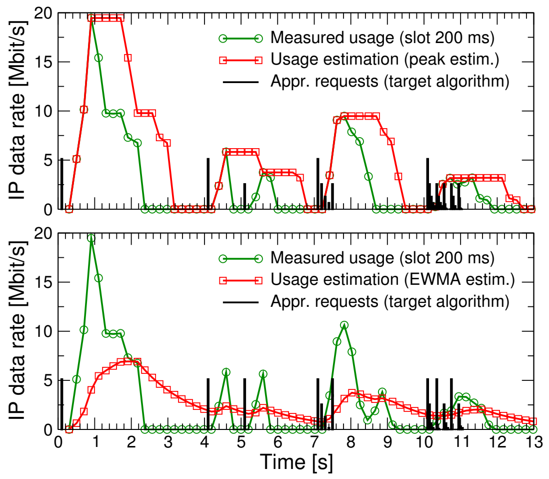


Figure 6.27: Quick-Start approval control with the target algorithm using the peak or EWMA rate estimator

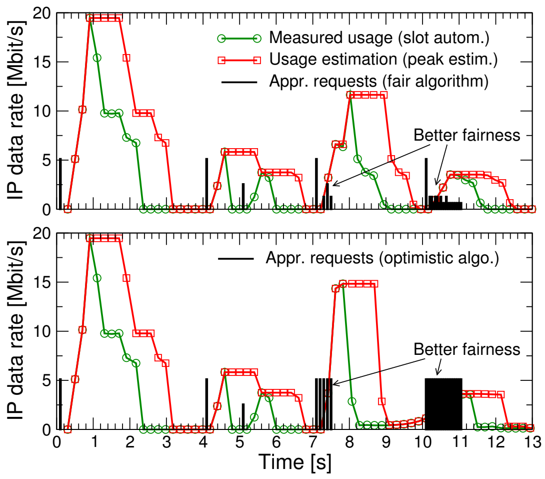


Figure 6.28: Quick-Start approval control with the fair or optimistic algorithm in combination with the peak rate estimator

be approved, which is confirmed by Figure 6.25. With cross-traffic, the throughput measured by the router differs from the one that is observed in the client. The upper part of Figure 6.26 reveals that router 1 indeed measures a carried traffic of 30 Mbit/s when the first Quick-Start request of 82 Mbit/s arrives. Due to the admission threshold of 100 Mbit/s, it can only approve 41Mbit/s. As a consequence, the data transfer starts with this rate.

The second request can be fully approved. The received sequence numbers are plotted in the lower part of Figure 6.26. They confirm that the second transfer starts only with the small restart window. The Quick-Start request is piggybacked on first packet. The high-speed data transfer starts once the corresponding ACK with the Quick-Start response has arrived after one RTT.

During the third transfer, the output link of router 1 is idle. Therefore, router 1 approves and allocates a rate of 82 Mbit/s. However, the admission control at router 2 denies the request, since the load on its outgoing interface exceeds the Quick-Start admission threshold. As a consequence, the third download starts in the Slow-Start phase, and router 1 spuriously reserves Quick-Start capacity during $\Omega = 2$ slots of duration $\Delta = 200\,\text{ms}$. This simple example shows that allocation in routers may be wasted due to lack of knowledge about the rest of the path.

As explained in Section 4.5.2, there are several degrees of freedom how to implement the approval control according to the target algorithm proposed in [192]. Two possible filters for the calculation of the link usage are a *peak estimator* and an *EWMA estimator*. Both alternatives are compared in Figure 6.27. These simulations assume a deterministic arrival pattern of Quick-Start requests that represent different request arrival patterns: One client-server transfer with a connection vector $\mathbf{V} = [(100, 2000000, 0)]$ starts at $t = 0\,\text{s}$. Two further mid-sized transfers with $\mathbf{V} = [(100, 200000, 0)]$ start at $t = 4\,\text{s}$ with a 1 s gap, then there are five transfers with $\mathbf{V} = [(100, 200000, 0)]$ at $t = 7\,\text{s}$, each with a 0.1 s gap, and finally the link is used by twenty smaller transfers with $\mathbf{V} = [(100, 20000, 0)]$ at $t = 10\,\text{s}$ with a 0.05 s gap. The traffic metering and approval control functions here operate with a constant control interval duration of $\Delta = \tau = 200\,\text{ms}$ and $\Omega = 2$. The traffic measurement is realized before the IP packets get queued in the bottleneck link with rate $r = 10\,\text{Mbit/s}$.

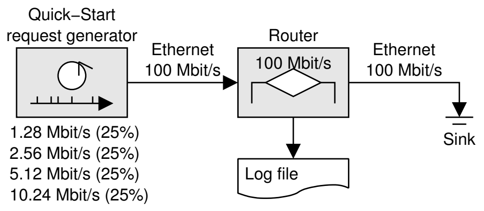


Figure 6.29: Quick-Start approval control load test with a mix of request rates

As Figure 6.27 shows, the peak estimator reacts instantaneously to an increase of the measured rate, and remembers a peak value during a duration of $N_{\text{peak}}\Delta = 1\,\text{s}$. As the router observes the *input traffic* to the bottleneck queue, the instantaneous rate may be larger than r. In contrast, the EWMA estimator reacts much more smoothly to sudden changes of the load. In the given example, the EWMA estimator, configured with $\alpha_{\text{ewma}} = 1/8$, grants more bandwidth to Quick-Start requests, but the overall difference between both techniques is small. This result is consistent with findings of Sarolahti *et al.* [192]. In this work the peak estimator is preferred due to its faster responsiveness.

6.3.3.3 *Performance of the proposed new approval control algorithms*

In Section 4.5.3.3, the *fair algorithm* and the *optimistic algorithm* are proposed as alternatives to the *target algorithm*. Figure 6.28 compares their characteristics in the same workload scenario like in the previous section. As to be expected, their behavior differs to the target algorithm if there are many competing requests. The upper part of Figure 6.28 shows that the fair algorithm reacts to the bulk arrival of requests at $t = 10\,\text{s}$. It indeed grants more or less the same, small rate to each of of the requests. In contrast, the target algorithm randomly grants a larger rate to selected ones, which is unfair. The lower part of Figure 6.28 reveals that the optimistic algorithm is even more fair: It justs approves all requests as long as the central link has spare capacity. But this also means that during the bulk arrival of requests at $t = 10\,\text{s}$ the available capacity is overbooked by one order of magnitude.

The benefits of the two alternative approval control algorithms can be observed more systematically in a load test without actual data traffic, which avoids interactions with the rate estimators. As depicted in Figure 6.29, the approval control can be challenged by a request generator that issues a large number of Quick-Start requests. In the practical setup, this Linux-based load generator has been realized by a program that generates raw IP packets carrying a Quick-Start request [226]. They have a configurable, exponentially distributed IAT. The maximum bandwidth available to requests is here set to $\theta \cdot c = 100\,\text{Mbit/s}$.

In theory, the target algorithm can grant a maximum bandwidth per second that is equal to

$$\Xi = \frac{\theta \cdot c - u}{\Delta \cdot \Omega}. \tag{6.1}$$

In this term, $\theta \cdot c - u$ is the available bandwidth and $\Delta \cdot \Omega$ the duration of the bandwidth pooling. The dependency of Ξ on the duration of the control interval Δ is confirmed by the measurement results presented in Figure 6.30. It depicts the granted bandwidth per second as a function of the requested value, which depends on the IAT of requests. The figure considers four different setings for the slot duration Δ between 100 ms and 1 s. The larger the value of Δ (or, alternatively, Ω), the more *conservative* is the target algorithm, i. e., the less bandwidth is granted.

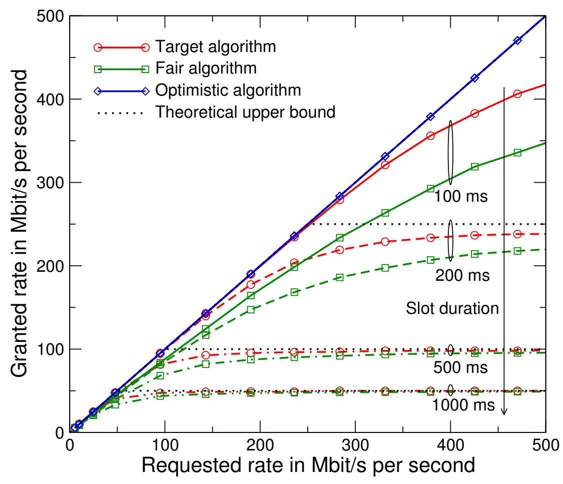


Figure 6.30: Efficiency of different approval control algorithms

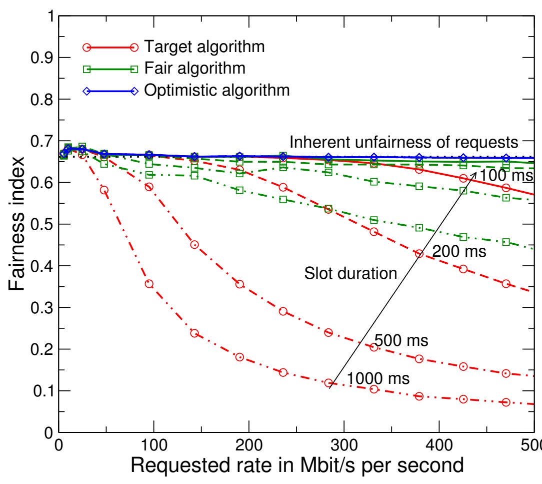


Figure 6.31: Fairness of different approval control algorithms

Figure 6.31 analyzes more in detail how the rate is allocated to the different requests. The chart has been obtained by calculating the *fairness index* of the approved rates according to Equation (4.2). Of course, the optimal situation is that all requests get approved. In the given setup, this outcome corresponds to a fairness index $FI < 1$, as the rate requests have different values. The result for an approval control with the target algorithm is much below this optimal value: If Δ is large, e. g., 500 ms or 1 s, the fairness index FI for the target algorithm is very small. This experiment proves that the target algorithm is indeed very unfair if resources are scarce and requests have to be reduced.

Figure 6.31 confirms that the fairness is significantly improved by the proposed alternative algorithms. As the optimistic algorithm approves all requests in this setup, the fairness index FI is equal to the theoretical optimum value. In other words, the optimistic algorithm is inherently fair because it allows an oversubscription of the resources.

Figure 6.31 shows that the fair algorithm achieves a reasonable level of fairness, too, even though it uses the principle bandwidth pooling like the target algorithm. The fairness index FI is slightly smaller than the optimum value. This effect is caused by the parameter Θ, which still allows some unfairness in order to achieve efficiency (see Section 4.5.3.3 for details). Furthermore, the fair algorithm only assigns equal rates if the arrival pattern of the requests is rather regular, which is not the case in this setup – and also very unrealistic in reality. Still, the achieved fairness is much better than the one of the target algorithm. The drawback of the fair algorithm is that less bandwidth is granted in total, which can be observed in Figure 6.30. This underutilization of the total size of the bandwidth pool is a side-effect of reserving some bandwidth for future requests, which may not always be used.

These example results indicate that there is a fundamental design trade-off between *high efficiency*, *high fairness*, and *low risk of congestion*, and that no approval control is optimal with respect to all three design goals. The target algorithm is efficient and conservative, i. e., it does try to minimize the risk of congestion. Both proposed new algorithms improve the fairness compared to the existing target algorithm – at the cost of either reducing the efficiency or increasing the probability of congestion due to resource oversubscription. The application performance implications of this trade-off is also investigated in Section 6.4.5.2. The experiments there will show that an approval control with oversubscription performs quite well in many situations.

6.4 Quantification of the potential performance improvement

6.4.1 New analytical model for fast startup performance

6.4.1.1 Existing models for short and mid-sized TCP data transfers

The potential performance improvement of fast startup schemes depends on the Slow-Start performance and on the specific realization of the fast startup. There are various closed-form analytical models for the transfer time of a given amount of data in Slow-Start. These existing Slow-Start models can be subdivided into two different classes that differ in their assumptions.

First, TCP's performance in Slow-Start can be limited by a maximum allowed Congestion Window, a limited initial Slow-Start Threshold, a small receiver advertised window, or by packet loss. In all these cases, the maximum throughput may remain below the theoretical maximum given by the available bandwidth on the path. Two well-known models for the influence of lost packets during Slow-Start can be found in references [42, 215].

Second, the Slow-Start algorithm imposes a fundamental performance limit. Even if a source can indeed send with a data rate equal to the available bandwidth on the path, the Slow-Start results in an additional delay until this optimal operation point is reached. Because of the exponential increase of the CWND, this delay depends on a logarithmic function of the BDP. This effect is obvious and has also been quantified in several publications [185, 20]. Bodamer [24] has extended this previous work and has derived a closed-form expression for file transfer times in Slow-Start. In the following, an adapted version of Bodamer's model is used to quantify the maximum possible improvement of fast startup congestion control.

Despite numerous related work on fast startup schemes, the exact performance improvement has hardly been calculated analytically. As an exception, Sarolahti *et al.* [192, 191] present an analytical close-form equation for the performance improvement of the Quick-Start TCP extension. However, the equations in [192, 191] are not correct when the amount of data exceeds the BDP. Furthermore, that model completely neglects the impact of the rate pacing and the consequences of an initial sending rate that is smaller than the available bandwidth. Thus, it only provides a rough upper bound on the performance improvement by Quick-Start in some usage scenarios.

The analytical models derived in the following sections are much more precise and overcome these shortcomings. An early version for Quick-Start has been published in reference [199]. It has later been extended to handle other fast startup schemes as well [205].

6.4.1.2 Data transfer times with Slow-Start

In this section, Bodamer's model as presented in [24] is briefly recapitulated. As shown in Figure 6.32, it is assumed that the end-to-end path can be characterized by a TCP path capacity $R = \frac{L}{MTU} \cdot r$ and its minimum RTT τ. r is the data rate at IP layer, MTU is the maximum transmission unit, and L is the maximum segment size (MSS).

The TCP behavior in Slow-Start can be modeled by *rounds* or *mini-cycles* that start when the sender begins the transmission of a window of packets and that end when an ACK for one or more of these packets arrives [42, 185, 215, 20]. In each round $i \geq 1$, the sender sends as many data segments as its CWND $W(i)$ allows. Depending on the usage of delayed ACKs, the receiver typically sends an ACK after having received η segments. Since the sender increases its CWND by one segment per ACK, the CWND at round $i+1$ follows as $W(i+1) \approx (1+\frac{1}{\eta})W(i) =$

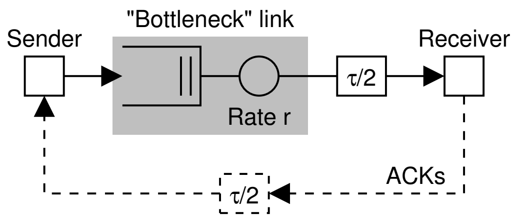


Figure 6.32: Simplified path model

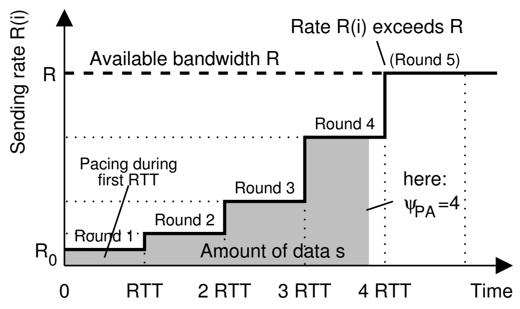


Figure 6.33: Fast startup with rate pacing

$\gamma \cdot W(i)$ with $\gamma = 1 + \frac{1}{\eta}$. With an initial window of w the window size in round i is $W(i) = w \cdot \gamma^{i-1}$. $W(i)$ corresponds to the maximum amount of data to be sent in that round. The amount of data transmitted in i rounds can be approximated by a geometric series:

$$M(i,w) = \sum_{j=1}^{i} W(j) = w \frac{\gamma^i - 1}{\gamma - 1}. \tag{6.2}$$

The maximum amount of data that is transferred in Slow-Start rounds is $M(\psi_{\mathrm{SS}}, w)$. ψ_{SS} is the index of the last Slow-Start round that is completely used by the sender. Once the sending window $W(i)$ exceeds the BDP, the throughput is limited by the data rate of the path. Under the assumption that the duration of a round is $T_{\mathrm{round}} = \tau + \frac{L}{R}$ and that it is independent of the window size, the transfer time of a given amount of data s in Slow-Start is

$$\Gamma_{\mathrm{SS}}(s,w) = \underbrace{T_{\mathrm{round}} \cdot \psi_{\mathrm{SS}}}_{\text{Delay by Slow-Start}} + \underbrace{\left(\frac{s - L \cdot M(\psi_{\mathrm{SS}}, w)}{R} \right)}_{\text{Transfer fully utilizing path}}. \tag{6.3}$$

This expression assumes that the initial value of the Slow-Start Threshold is larger than the BDP and that the connection is not receiver-limited. In order to calculate ψ_{SS}, two cases have to be considered: If the data transfer is long enough, the sender fully utilizes the pipe in round κ_{SS} if $W(\kappa_{\mathrm{SS}})$ exceeds the BDP, i. e., $W(\kappa_{\mathrm{SS}}) \geq R \cdot T_{\mathrm{round}}/L$. Solving for κ_{SS} gives:

$$\kappa_{\mathrm{SS}} = \left\lceil \log_\gamma \left(\frac{R \cdot T_{\mathrm{round}}}{w \cdot L} \right) \right\rceil + 1 \; . \tag{6.4}$$

Short transfers may not arrive at this point. If the amount of data is completely transferred in Slow-Start rounds only, the number of rounds ν_{SS} follows from $M(\nu_{\mathrm{SS}}, w) \geq \lceil \frac{s}{L} \rceil$:

$$\nu_{\mathrm{SS}} = \left\lceil \log_\gamma \left(\left\lceil \frac{s}{L} \right\rceil \frac{\gamma - 1}{w} + 1 \right) \right\rceil \; . \tag{6.5}$$

By definition, the last complete round has the index

$$\psi_{\mathrm{SS}} = \max\left(\min\left(\kappa_{\mathrm{SS}}, \nu_{\mathrm{SS}}\right) - 1, 0\right). \tag{6.6}$$

6.4.1.3 New model for rate paced fast startup schemes

One fundamental difference between the Slow-Start and most of the considered fast startup schemes is the use of rate pacing during the first RTT. Due to *ACK clocking*, rate pacing also changes the subsequent traffic pattern, since the segments are not all sent at once, but equally spaced over an RTT. Because of this difference, Equation (6.3) does not correctly describe a data transport that starts with rate pacing. Rate pacing can correctly be analyzed by a new model [205] that modifies Bodamer's equations accordingly.

With rate pacing, the sender approximately uses an initial sending rate Q during the first RTT. If the sender uses the Slow-Start algorithms after the initial playout of data, i. e., if it does not switch to Congestion Avoidance, the sending rate is further increased. This means that again *rounds* with a duration T_{round} occur, as depicted in Figure 6.33. However, unlike in the model in the previous section, data is continuously sent within the round. As long as the available TCP path capacity R is not reached, the sending rate is increased by a factor $\gamma = 1 + \frac{1}{\eta}$ every round, i. e., data is continuously sent with rate $Q(i) = Q \cdot \gamma^{i-1}$ during round i. Therefore, one must distinguish like in Equation (6.3) a sending phase in which the rate $Q(i)$ is smaller than R, and another one in which the path is fully utilized. The transfer time then follows as

$$\Gamma_{\text{PA}}(s,Q) = T_{\text{round}} \cdot \psi_{\text{PA}} + \frac{s - M(\psi_{\text{PA}}, Q \cdot T_{\text{round}})}{\min\left(R, Q \cdot \gamma^{\psi_{\text{PA}}}\right)}. \qquad (6.7)$$

In this expression, $M(i,w)$ is the maximum amount of data that can be sent in round i. It is given by Equation (6.2). ψ_{PA} is again the index of the last round with a rate $Q(i) < R$. Analogous to Equation (6.6), it depends on two different effects and is defined as

$$\psi_{\text{PA}} = \max\left(\min\left(\kappa_{\text{PA}}, \nu_{\text{PA}}\right) - 1, 0\right). \qquad (6.8)$$

The first term with κ_{PA} refers to the index of the round in which the available path capacity R is reached. ν_{PA} is the round in which all data would have been sent. The corresponding values differ from Equation (6.4) and Equation (6.5) in several details:

$$\kappa_{\text{PA}} = \left\lceil \log_\gamma\left(\tfrac{R}{Q}\right)\right\rceil + 1 \qquad (6.9)$$

$$\nu_{\text{PA}} = \left\lceil \log_\gamma\left(\tfrac{s \cdot (\gamma - 1)}{Q \cdot T_{\text{round}}} + 1\right)\right\rceil. \qquad (6.10)$$

6.4.1.4 Total transfer times

Equation (6.3) and Equation (6.7) only quantify the transfer time. Latencies on the path have to be considered separately. For instance, if the transfer time of a response is Γ, and if the request message size can be neglected, the minimum response time of a client-server application follows as $T_{\text{resp}} = \Gamma + \tau$. If a connection has to be established first, an additional RTT is required for the three-way handshake, which results in a total delay of $2\,\tau + \Gamma$.

For different startup schemes, Γ depends on the parametrization. Table 6.1 lists the corresponding expressions for the algorithms that are used in the following experiments. The minimum server response time of Initial-Start follows directly from Equation (6.3) with a larger initial window w_{IW}. In case of Jump-Start, the initial rate Q depends on the amount of queued application data s and the thresholds K_{data} and K_{rate}. Mega-Start and Quick-Start both depend on the used initial rate Q. However, in case of Quick-Start there is only a limited number of valid values for Q [RFC 4782]. Furthermore, the signaling delay of one RTT must be taken into account depending on the Quick-Start usage scenario.

Table 6.1: Response time equations for different startup schemes

Scheme	Additional parameters	Minimum response time $T_{\mathbf{resp}}(s)$
Linux Slow-Start	–	$\tau + \Gamma_{\mathrm{SS}}(s, w)$
Initial-Start	Initial window w_{IS}	$\tau + \Gamma_{\mathrm{SS}}(s, w_{\mathrm{IS}})$
Jump-Start	Thresholds K_{data} and K_{rate}	$\tau + \Gamma_{\mathrm{PA}}(s, \min(s/\tau, K_{\mathrm{data}}/\tau, K_{\mathrm{rate}}))$ if $\min(s, K_{\mathrm{data}}, K_{\mathrm{rate}} \cdot \tau) > w \cdot L$
Mega-Start	Initial rate $Q = \frac{L}{MTU} \cdot q$	$\tau + \Gamma_{\mathrm{PA}}(s, Q)$ if $Q \cdot \tau > w \cdot L$
Quick-Start (early activ.)	Approved rate $Q = \frac{L}{MTU} \cdot q$	$\tau + \Gamma_{\mathrm{PA}}(s, Q)$ if $Q \cdot \tau > w \cdot L$
Quick-Start (late activ.)	Approved rate $Q = \frac{L}{MTU} \cdot q$	$2\tau + \Gamma_{\mathrm{PA}}(s - w, Q)$ if $Q \cdot \tau > \gamma \cdot w \cdot L$

The parameters in these analytical models are as follows: In the Linux network stack the MSS is $L = 1448\,\mathrm{B}$ for an MTU of $1500\,\mathrm{B}$. The "QuickAck" mechanism during a Slow-Start results in $\eta = 1$. Linux's initial CWND in Linux kernel version 2.6.18 depends on the scenario: At the beginning of a connection it is $w = 2$, whereas $w = 3$ is used after long idle times. Apparently, the motivation for this difference is to avoid interactions with erroneous implementations of delayed acknowledgments.

6.4.2 Comparison of the potential speedup

6.4.2.1 Performance improvement under ideal constraints

If the assumptions of the analytical model are fulfilled, the performance benefit of fast startup schemes depends on the amount of data s, the available bandwidth r, and the minimum RTT τ. In the following, the possible speedup of data transfers is studied under such ideal constraints, and the protocol implementations are validated against the analytical models.

The setup again consists of a client and a server. The communication is described by a connection $\mathbf{V} = [(100, s, 0)]$, which is transported over a new TCP connection. s is varied over several orders of magnitude. The experiment thus corresponds to the communication pattern shown in Figure 6.21. Both in simulations and measurements, the link capacity is $r = 10\,\mathrm{Mbit/s}$. The buffer size is set to $B = 1000$ packets to enable similar constraints for simulation and measurements. In order to illustrate the fundamental effect, the minimum RTT is assumed to be 200 ms in the first diagrams. Afterwards, the RTT as well as be bandwidth are varied. All presented results have been obtained with the CUBIC congestion control. Other experiments have revealed that there are only few differences if the Reno algorithms is used; they are published in [204].

The main performance metric is here the *server response time* T_{resp}, which is the duration between the completion of the three-way-handshake and the end of the data transfer. Without random cross-traffic, the simulation result for the response time is deterministic, i. e., confidence intervals are not required. For the testbed measurements, each presented response time value is the average of ten consecutive measurements.

Figures 6.34–6.37 print the server response time T_{resp} as a function of the response size s, i. e., the amount of data sent by the server. All figures compare the analytical prediction for the response times with the simulation and testbed measurement results, i. e., there are three graphs for each considered flow startup mechanism. The graphs for standard TCP, which is shown in all figures, reveals the typical steps of the Slow-Start. These steps occur whenever a new Slow-Start round is required. The Slow-Start of the simulated network stack is exactly modeled by

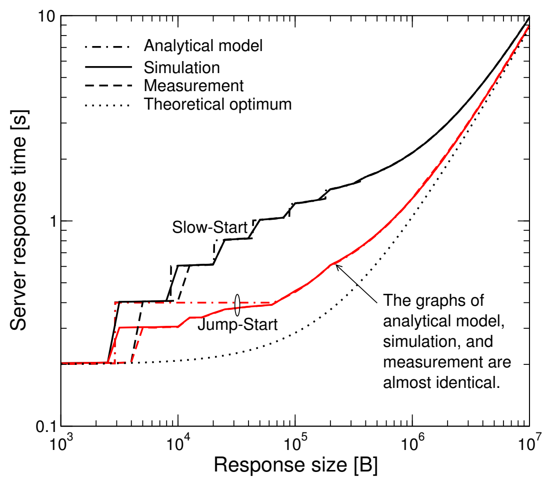


Figure 6.34: Benefit of Jump-Start as a function of the transfer size

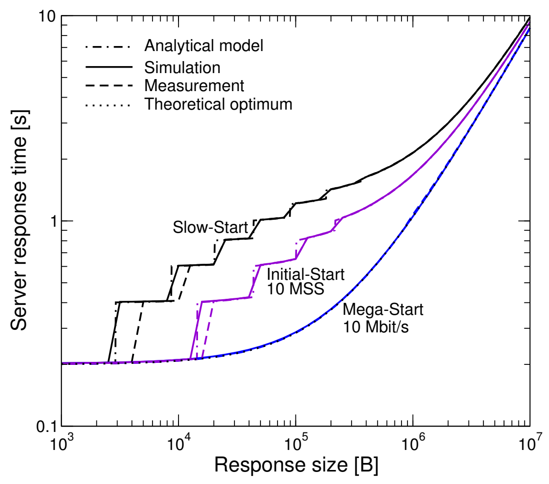


Figure 6.35: Benefit of other end-to-end schemes as a function of the transfer size

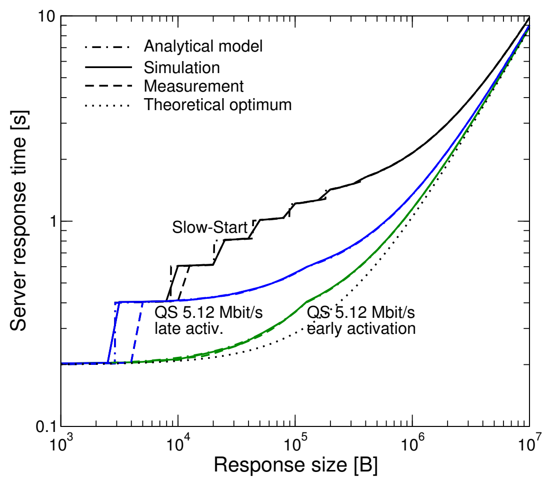


Figure 6.36: Benefit of Quick-Start as a function of the transfer size

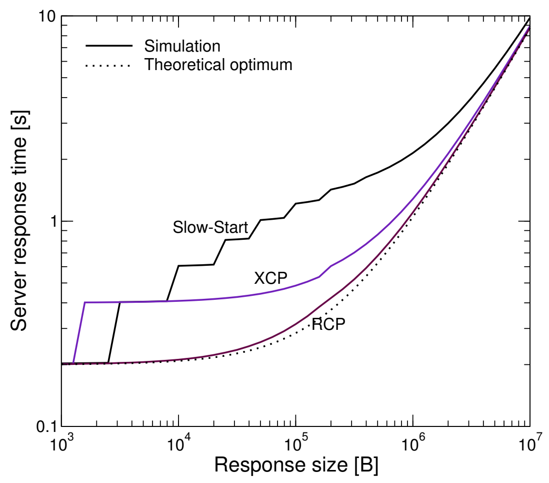


Figure 6.37: Benefit of network-controlled schemes as a function of the transfer size

Equation (6.3). However, there is a difference between the simulation and the measurement. As mentioned in Section 6.1.1.2, the Slow-Start implementations in Linux kernel versions 2.6.18 and 2.6.24 are not identical and differ in the initial window. Only the kernel that is used in simulations uses an initial window of $w = 2$. But the difference between both versions is only significant for transfers up to 10 kB. Above this value the simulation results, the measurements, and the analytical model match very well.

Fast startup schemes can be expected to improve the response time in particular for mid-sized response sizes of the order of several dozens of kilobytes [192], i. e., transfers between "mice" and "elephants". The charts in Figures 6.34–6.37 confirm this is inherent characteristic of all fast startup scheme. Though, there are specific differences.

As shown in Figure 6.34, Jump-Start results in substantially faster data transfers if the available data exceeds the initial window w. As Jump-Start is designed to play out the available application data during one RTT τ, the response time should ideally be given by $T_{\mathrm{resp}} = 2\,\tau$ for $w < s \leq K_{\mathrm{data}}$. However, Figure 6.34 reveals that the actual response time of the real implementation is smaller for $s \ll K_{\text{data}} = 64\,\text{KiB}$. As explained in Section 5.3.2.2, rate pacing can in practice only be realized with a limited number of timers. If s is small, only two timers make sense. But with such a small number of timers it is impossible to realize an ideal rate pacing.

Other end-to-end schemes can result in a much larger initial sending rate. If the initial window is just increased to a larger value $w_{\text{IS}} = 10$, bursts up to this size can be sent out immediately (Initial-Start). The resulting graph, which is depicted in Figure 6.35, basically represents a vertical shift of the Slow-Start graph. Figure 6.35 also shows the theoretical optimum that would be achieved if the available bandwidth on the path $r = 10\,\text{Mbit/s}$ was somehow known and used (Mega-Start).

In case of Quick-Start, the performance benefit depends on the activation method. According to Figure 6.36, if either the application or the kernel-internal heuristics initiate a request during the three-way handshake (*early activation*), Quick-Start significantly speeds up transfers in the range of few kilobytes to about one megabyte. As long as $s < Q \cdot \tau$, the sender can play out all data with the approved rate, which is $q = 5.12\,\text{Mbit/s}$ in the given example. Larger transfers completely utilize the path after some time. In the Quick-Start graphs in Figure 6.36 there is also a small kink at $s = 128\,\text{kB}$, which corresponds to the amount of data that can completely be sent during the rate pacing phase. If Quick-Start gets activated once application data is indeed available (*late application*), one additional RTT is required for the signaling. This delay results in a smaller performance benefit, which is also shown in Figure 6.36. The simulation and measurement results again correspond to target value given by the analytical models in Table 6.1.

The fundamental difference between the philosophy of *bandwidth pooling* and *oversubscription* can be observed in Figure 6.37. In XCP, the RTT must be known by the network components before bandwidth out of the resource pool is granted to a flow [64]. As a consequence, XCP requires an additional RTT before it can start to send with a high sending rate. The initial window allowed by XCP without positive feedback is not specified and here assumed to be one MSS. The performance is thus similar to Quick-Start with late activation. In contrast, RCP permits bandwidth oversubscription. As a result, the available bandwidth can immediately fully be utilized and the response time is almost identical to the theoretical minimum.

6.4.2.2 Impact of the RTT and the available bandwidth

The benefit of fast startup congestion control fundamentally depends on the RTT of the path. Figures 6.38 and 6.39 depict as examples the relative improvement of Jump-Start and Quick-Start for different RTTs. This improvement is calculated from the ratio of the server response time with Slow-Start divided by the one with the fast startup according to Table 6.1.

Both diagrams reveal a characteristical pattern: For a given RTT, the fast startup is only of limited benefit for small amounts of data, since such transfers can be completed in a few RTTs anyway. Long bulk data transfer are also not significantly improved, because the Slow-Start is only a transient phase and has only marginal impact on the overall download time. But there is a benefit for *mid-sized transfers* in between, i. e., for data volumes between 10 kB and 1 MB. In this range, the Slow-Start has a delaying effect. If the RTT is small, the effect is only small. But for RTTs of the order of 100 ms or larger, *server response times can be improved by several hundred percent*. The larger the RTT, the higher the potential speedup. These diagrams confirm similar empirical findings of Sarolahti *et al.* for Quick-Start [192].

When the available bandwidth on the path increases, the relative performance improvement can be much larger. This effect can be observed in Figure 6.40, which shows simulation results for

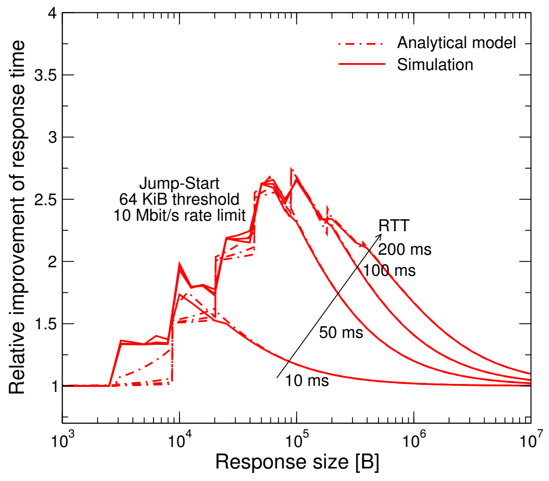


Figure 6.38: Relative performance improvement of Jump-Start as a function of the RTT

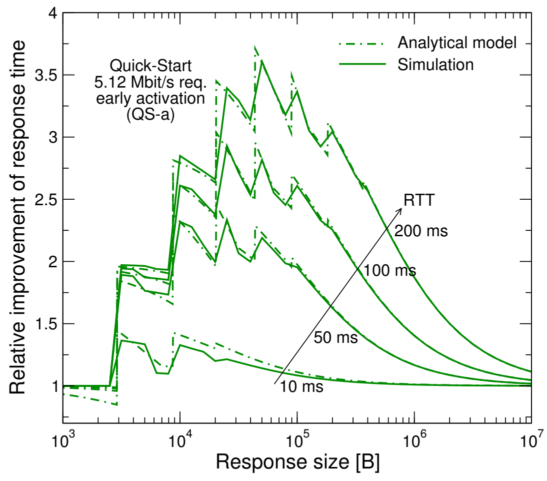


Figure 6.39: Relative performance improvement of Quick-Start as a function of the RTT

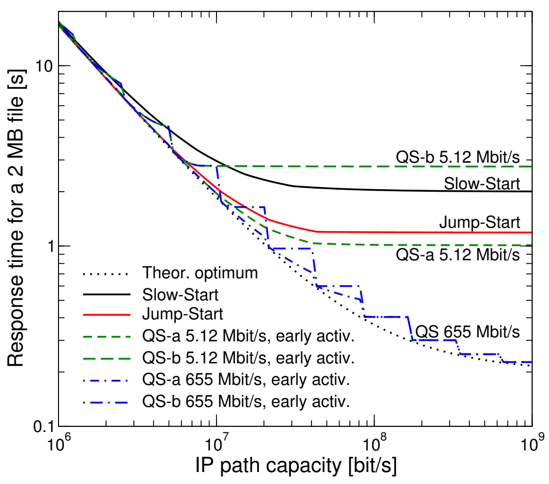


Figure 6.40: Benefit of selected schemes for higher path capacities

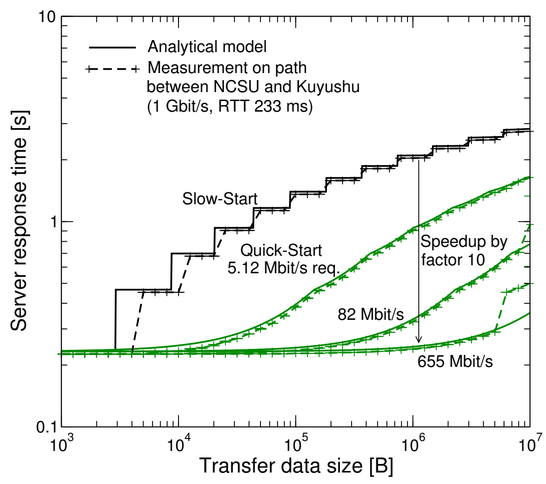


Figure 6.41: Measured speedup on a real 1 Gbit/s path with different QS request rates

such a path with an RTT of 200 ms. A mid-sized data transfer in Slow-Start cannot utilize the bandwidth if it exceeds several Mbit/s. The minimum download time is of the order of 2 s, even if the path capacity is very large. In contrast, the theoretical optimum can almost be reached if a sufficiently large Quick-Start request is issued and granted. This corresponds to a speedup of one order of magnitude. There are certain steps in the Quick-Start graph, which correspond to the few possible Quick-Start request rates. If only a smaller rate of $q_{\text{req}} = 5.12$ Mbit/s is requested, the path is not completely utilized, and the response time can only be improved by about one second.

In this setup, Jump-Start also improves the delay by about one second only and cannot completely benefit from the large available bandwidth. Besides this, Figure 6.40 shows once again that a reduction of the SST after the Quick-Start phase is not optimal and that the QS-b strategy can even result in a worse performance than Slow-Start.

These results have also been confirmed on a real long-distance path with a data rate of $r \approx$ 1 Gbit/s. Figure 6.41 presents the results of an experiment with different Quick-Start request rates. Due to lack of full Quick-Start router support, this study assumes that requests up to that rate are indeed approved on the path, since the path was not significantly loaded by other traffic. The Quick-Start mechanism can achieve substantial *transfer time speedups up to a factor of ten* in such a high-speed environment, if the endsystem asks for a sufficiently high rate. This means that the transport delay are reduced by up to several seconds, which is perceivable by a human user.

6.4.3 Bandwidth sharing properties

6.4.3.1 Implications of cross-traffic

A flow startup scheme is challenged if a large number of flows arrive on a bottleneck link. In order to study this important situation, both simulations and measurement have been performed for a client-server communication between $N_{\text{hosts}} = 200$ client-server pairs. For simplicity, it is assumed that client and servers exchange requests and responses over $N_{\text{hosts}} = 200$ persistent TCP connections. In this case, Slow-Start is also used since the *Congestion Window Validation* [RFC 2861] reduces the CWND during the idle times to the *restart window* (in case of Linux, $w = 3$). In order to adjust a certain load ρ, random request-response vectors are scheduled on these persistent TCP connections. The requests are small (100 B), and the response length is obtained from a *truncated pareto distribution* with mean $m = 114\,\text{kB}$, shape factor $\alpha_{\text{pareto}} = 1.1$, and a cutoff at $K_{\text{pareto}} = 10\,\text{MB}$. This distribution is introduced in Section A.1. This workload model thus uses a similar response size distribution like the traffic measurements presented in Section 3.3.3.2, but assumes larger average transfer sizes, i. e., a traffic scenario in which more mid-sized transfers occur. Experiments that reflect the current Internet workload characteristics are presented in the next section. The IAT of vectors is exponentially distributed with mean $\delta = m/R/\rho$, which is a realistic assumption according to Section 3.3.3.2. The parameters of the dumb-bell topology are $r = 10\,\text{Mbit/s}$ and $\tau = 200\,\text{ms}$. In the simulations, the central bottleneck has a reasonable buffer size of $B = 50$ packets.

The theoretical minimum average server response time for a lightly loaded bottleneck ($\rho \ll 1$) can be calculated by integrating over the response size probability distribution (cf. Section A.1):

$$T_{\text{resp,min}} = \int_{k_{\text{pareto}}}^{K_{\text{pareto}}} f_{\text{tpareto}}(z) \cdot T_{\text{resp}}(z)\,\mathrm{d}z. \tag{6.11}$$

The corresponding expressions for $T_{\text{resp}}(z)$ as a function of the amount of data z depend on the flow startup scheme and are given in Table 6.1. When the load ρ increases, it is hardly possible to exactly calculate the server response time, as the link is not equally shared among the data transfers. Furthermore, packet losses will occur, but their influence cannot be described by a simple model. Yet, it is possible to model an *ideal case* without the impact of TCP's protocol mechanisms, in which all ongoing transfers equally share the bottleneck link. The response time in this case can be determined from an M/G/1 *Processor Sharing* (PS) model [20]:

$$T_{\text{resp,ps}} = \tau + \frac{m}{R\,(1-\rho)}. \tag{6.12}$$

If the individual rate of the flows was limited, this model could be extended by an M/G/r PS model [185, 20].

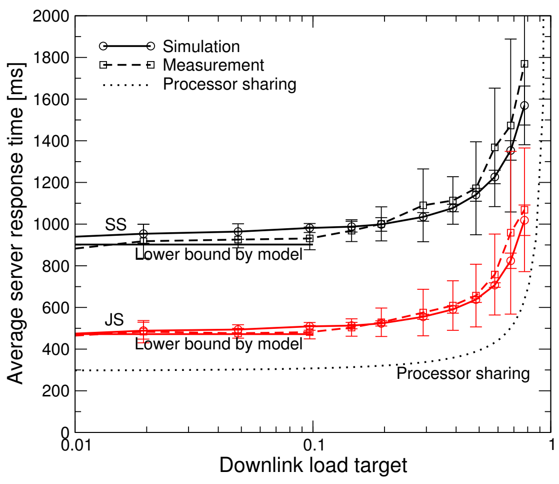


Figure 6.42: Impact of an increased load on Jump-Start

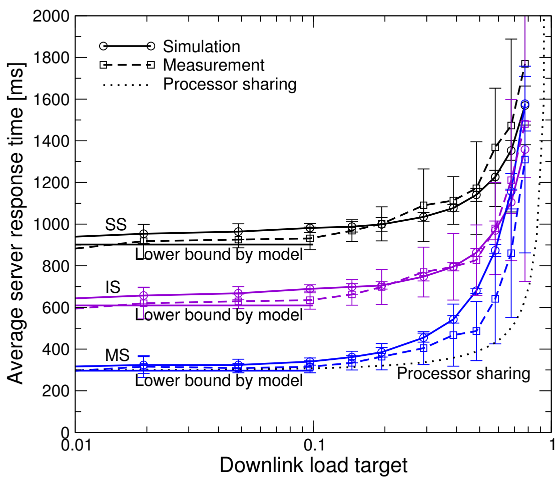


Figure 6.43: Impact of an increased load on other end-to-end schemes

6.4.3.2 Comparison of the different fast startup schemes

Figure 6.42 compares the performance of Slow-Start and Jump-Start scheme in this setup, both with measurement and simulation data. Obviously, the response time increases as the load ρ at the bottleneck gets larger. Also, the response times have then a much larger variance. In average, the Jump-Start scheme can improve the server response time by several hundred milliseconds in this experiment. It does not reach the ideal case of processor sharing according to Equation (6.12), which is also shown in Figure 6.42, but the difference is rather small. Jump-Start can also keep the response time at a low level even if the bottleneck load is high. This indicates a reasonable behavior of the Jump-Start approach.

As shown in Figure 6.43, other end-to-end scheme have a similar performance as long as the bottleneck load is small. But there are differences once the load increases. If $\rho \gg 0.1$, both the Mega-Start and the Initial-Start schemes are not significantly better than the Slow-Start. In the former case, packet losses are very likely to occur if each flow starts approximately with full link speed. In the latter case, the lack of pacing increases the risk of buffer overflow. If the initial window window was larger, i. e., $w_{\mathrm{IS}} > B$, the performance of Initial-Start would be even worse than Slow-Start [205].

In the given scenario with persistent connections, Quick-Start can only use the *late activation*, i. e., it needs one RTT for the signaling. This means that the optimal speedup compared to other schemes is smaller. The graph shown in Figure 6.44 has been obtained with the target algorithm approval control with $\Delta = 200\,\mathrm{ms}$ and $\theta = 1$. This means that requests are approved if the resulting bandwidth, including the current traffic and recently approved requests, is less than the bottleneck link capacity. Once the load ρ increases, i. e., when many requests arrive in parallel, less requests are approved, or the bandwidth granted to each request is smaller. In case of a very high load almost all requests are denied. This effect can be seen in Figure 6.44: For a high load, the Quick-Start protocol has more or less the same performance like a flow start with the standard Slow-Start.

The corresponding results for XCP and RCP are depicted in Figure 6.45 (with $B \longrightarrow \infty$). This diagram again shows that XCP is ineffective for short- and mid-sized transfers and that it may even be slower than TCP's Slow-Start if there are several competing flows. RCP indeed closely

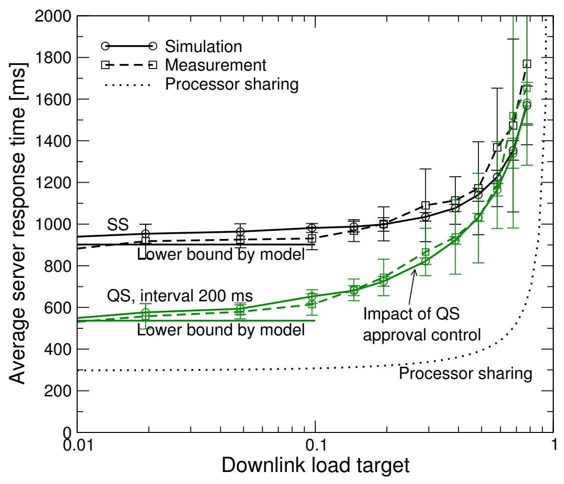


Figure 6.44: Impact of an increased load on Quick-Start

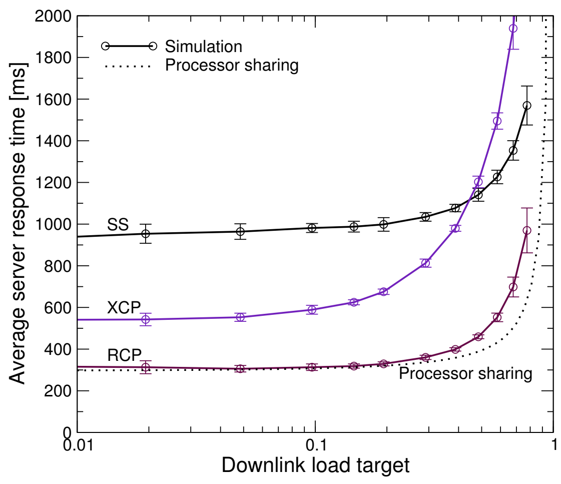


Figure 6.45: Impact of an increased load on XCP/RCP

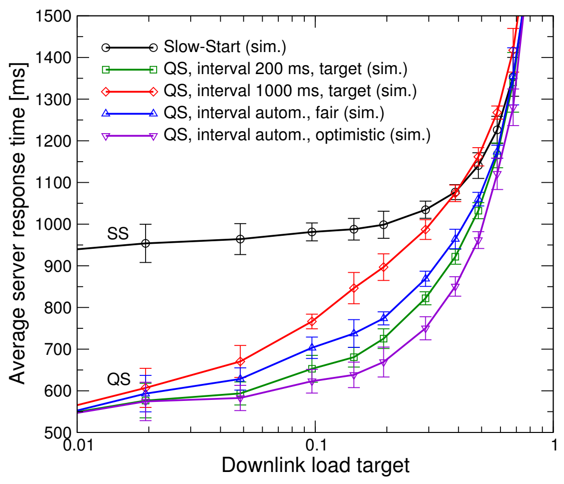


Figure 6.46: Load dependency of different QS approval control algorithms

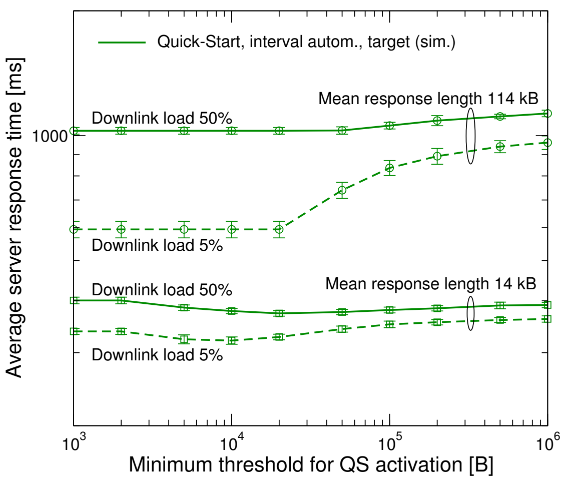


Figure 6.47: Benefit of activating Quick-Start only for larger transfers

approximates processor sharing. These results as well as similar studies in [181] thus confirm the findings of Dukkipati [57] with a completely independent simulation tool implementation of XCP and RCP.

6.4.3.3 Benefit of the intelligent usage of Quick-Start

On moderately loaded links, the performance of Quick-Start can be significantly optimized by improved algorithms. As introduced in Section 4.5.2, the target admission control algorithm remembers requests for a time duration of $\Omega \cdot \Delta$. It has first been observed by the author in [199] that the number of slots Ω and control interval duration Δ have a significant influence on the probability that a request is approved.

In order to illustrate this impact, selected combinations of approval control algorithms are compared in Figure 6.46: The number of slots is left constant ($\Omega = 2$), but different strategies are used to determine Δ: The interval duration is either set to constant values of 200 ms or 1000 ms, or automatically adapted as proposed in Section 4.5.3.4. If the bottleneck load is rather small, the interval duration has hardly any impact. But as the load increases, the approval control grants much less requests if Δ is larger.

This dependency on the duration control interval can easily be understood by a simple approximation of the approval probability of a Quick-Start request: On the one hand, the maximum number of requests that can be approved within the history duration $\Omega \cdot \Delta$ is $c \cdot \theta \cdot (1-\rho_{\text{estim}})/q$. Therein, ρ_{estim} is the estimated link load, and the threshold θ gives the ratio of capacity available for Quick-Start requests. On the other hand, the number of requests within the history duration is Poisson distributed with mean $\Omega \cdot \Delta/\delta$. The probability that a request gets approved can be approximated by dividing the number of approvable requests by the total number of requests during the slot duration, which results in a dependency $p_{\text{approv}} \propto (1/\rho - 1)/(\Omega \cdot \Delta)$. This term is an approximation only, since the instantaneously measured load used by the approval algorithms is not identical to the average load. But it illustrates that $p_{\text{approv}} \approx 0$ if either $\rho \rightarrow 1$ or if $\Omega \cdot \Delta$ is large. This is why the setting of the control interval Δ is crucial.

Figure 6.46 also shows that an additional fairness enforcement as proposed in Section 4.5.3.3 is more conservative and grants less capacity than the target algorithm. This is the expected behavior, since the *fair algorithm* always keeps some unallocated bandwidth for future requests.

The opposite solution is the *optimistic algorithm*, which does not store information about previous requests and thus oversubscribes bandwidth (cf. Section 4.5.3.2). This approval control grants more requests and results in a speedup in particular on a moderately loaded link, i. e., if ρ is of the order of 30 %. For very small and very high loads, there are no significant differences between the different approval control strategies, as either all or no requests get approved. This result indicates that the optimistic algorithm does not overly increase the congestion risk.

Due to the bandwidth pooling principle, the target algorithm is not optimal if many Quick-Start requests are issued for transfers that are not able to send with the approved rate for at least some RTTs. The solution to this problem on the network-side is the *optimistic algorithm*. However, an even better approach is to avoid useless Quick-Start requests. This can be realized by an *intelligent activation* strategy that is proposed in Section 6.4.3.3.

The implications of this intelligent activation are shown Figure 6.47: The graphs in the upper part of the diagram correspond to the previously used workload model with mean $m = 114\,\text{kB}$. If the threshold χ introduced in Section 4.5.3.5 is small, all transfers use Quick-Start. This is the optimal case for this type or workload, where most transfers are large enough to reasonably utilize the granted Quick-Start rate. Obviously, it does not make sense to set χ to a very large value ($\chi \gg 100\,\text{kB}$), since then only very few long transfers use Quick-Start without a significant benefit.

The usefulness of the intelligent activation is more evident if the average transfer size is smaller. In the lower part of Figure 6.47, the response sizes are obtained from a truncated pareto distribution that is parametrized to match the real traffic characteristics of the traces, as introduced in Section 3.3.3.2, i. e., $m = 14\,\text{kB}$, $\alpha_{\text{pareto}} = 1.1$, and $K_{\text{pareto}} = 10\,\text{MB}$. In this case, there are many small transfers, too. Figure 6.47 shows that the average download time can be improved by setting χ to a value of the order of 10 kB, i. e., Quick-Start requests are only triggered if the server's response is larger than 10 kB. A value that is either orders of magnitude larger or smaller does not result in better performance. The impact of the intelligent activation mechanism would be even larger if a workload model with more small transfers was used. In the following sections, the default threshold is $\chi = 10\,\text{kB}$.

Table 6.2: Important parameters of the used Web workload models

Parameter	**"SURGE default" model**	**"Large files" model**
Ratio of base/embedded/loners files	0.30 / 0.38 / 0.32	0.30 / 0.38 / 0.32
File popularity	Zipf, 2000 files	Uniform, 2000 files
File size distribution	Mixed lognormal and pareto	Lognormal, $\mu = 11.92$, $\sigma = 1.0$
Number of embedded files per object	Pareto, $\alpha = 1.245$, $k = 2$	Geometric, mean value 2
User think time	Pareto, $\alpha = 1.4$, $k = 2$	Pareto, $\alpha = 1.4$, $k = 2$

Legend: α, k and μ, σ are the chacteristical parameters of the corresponding distribution

6.4.4 Studies with synthetic source-level Web models

6.4.4.1 Scenario and models

In the previous sections, only simple workload models have been considered in order to study the principal behavior. The performance of interactive applications can be more realistically investigated by experiments with real HTTP downloads from a Web server. Realistic request patterns can be generated here by a SURGE Web traffic load generator. The SURGE tools, which are introduced in Section 3.3.3.1, support HTTP/1.0 and HTTP/1.1 requests without or with pipelining. They are used in the default configuration, i. e., there is a configurable number of users/threads that each send HTTP requests to the Web server. Similar to the previous experiments, the minimum RTT is $\tau = 200\,\text{ms}$ and the emulated path capacity $r = 10\,\text{Mbit/s}$. The performance metric of interest is the total page download time T_{page}, which consists of a base document and potentially also embedded files (see Figure 3.3). In this experiment the measurement duration is one hour per sample.

Concerning the object characteristics and request patterns, both the default and a customized workload model are used. The important parameters of both models are listed in Table 6.2. The SURGE model [15] represents typical Web characteristics about one decade ago, and most files are rather small. But it is still a standard model for Web performance evaluations. In order to demonstrate the influence of larger object sizes, a second, hypothetical workload model is used, which has been introduced in [204]. This workload model is assumed to reflect the traffic characteristics of a broadband interactive application with frequent mid-sized data transfers. Therein, the mean object size is assumed to be 250 kB, which is consistent with some findings of the measurements in [209]. Other parameters have been set to a reasonable value.

6.4.4.2 Measurement results

Figure 6.48 presents the result of experiments with either a default network stack, Jump-Start, or Quick-Start. The other fast startup schemes are here omitted, since their performance impact is quite similar. For a rather low load of one simulated user, the mean page download time with Slow-Start is always larger than with Jump-Start or Quick-Start, independent of the HTTP protocol variant being used. However, the absolute difference is rather small for the "SURGE default" model. This is to be expected, since the mean size of an object (i. e., a Web page) is here of the order of 17 kB only. In contrast, for larger file sizes, both considered fast startup schemes can significantly reduce the download duration. A *speedup of more than* 1 *s* can be achieved if HTTP/1.0 is used, which transfers each object over a new connection with an initial Slow-Start. But a significant improvement can also be observed for HTTP/1.1 persistent connections, both

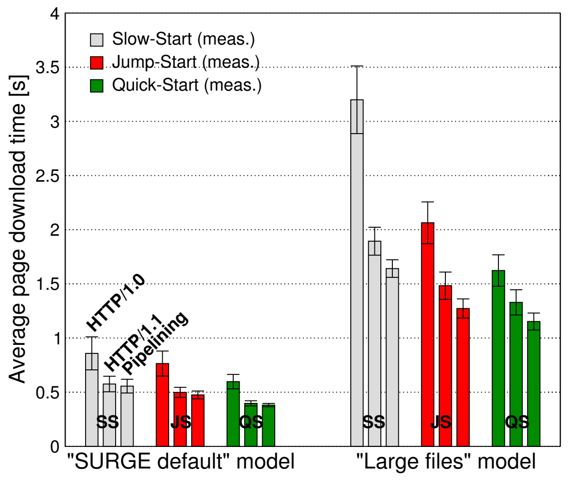


Figure 6.48: Comparison of the measured page download times for one user

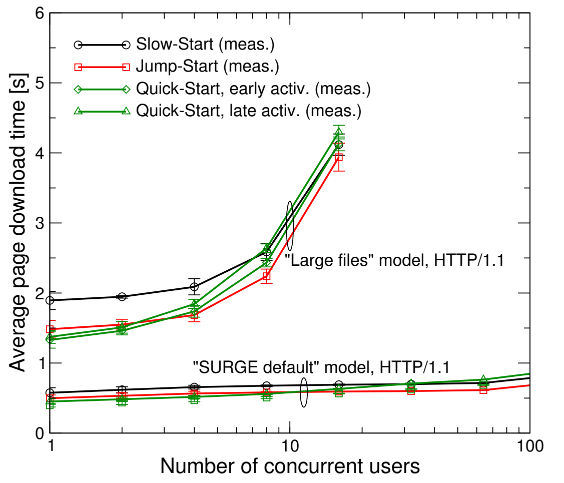


Figure 6.49: Measurement of the impact of an increased load

without and with request pipelining. To sum up, Figure 6.48 shows again that the performance benefit of fast startup schemes depends a lot on the characteristics of the Web pages. For small pages of the order of 10 kB, there is hardly any performance benefit. However, frequent transfers of the order of 100 kB or more can be significantly accelerated.

Figure 6.49 compares the page download durations for the two Web models when the load increases. In order to be more realistic, it only considers data transfers by the most commonly used HTTP version 1.1. It also distinguishes between early and late Quick-Start activation ($q_{\text{req}} = 5.12\,\text{Mbit/s}$). As to be expected, the download durations increase if the load gets larger. The result again reveals the fundamental difference between Quick-Start and end-to-end fast startup schemes: Quick-Start only improves the performance as long as the path is only lightly loaded. As soon as the load increases, the page download times are equal to Slow-Start or even worse, because less requests are approved and because the granted bandwidth is smaller. This illustrates that the Quick-Start mechanism mainly targets at underutilized links. In contrast, Jump-Start speeds up the page download times even if the load is high.

The actual performance of a Web application is hard to model by a simple setup, since it typically consists of queries to different servers with different path characteristics. Also, the question whether already established TCP connections can be reused, or not, depends significantly on the structure of the content and the application communication characteristics. In general, a fast startup is more beneficial if content is retrieved from several servers instead of one, if transfer sizes are larger, and if at least part of the content is retrieved from topologically far servers. Another example for the potential improvement for a Website is presented in Section 7.1.

6.4.5 Studies with trace-based workloads

6.4.5.1 Scenario and models

In order to quantify the performance benefit of fast startup schemes, realistic workload and delay models must be used. Specifically, connections with small RTTs must be considered, too. This can be achieved by replaying the real trace-based workloads introduced in Section 3.3.3.2 over the dumb-bell topology shown in Figure 6.3, which covers a whole range of RTTs. As already mentioned, this setup is recommended in the common TCP evaluation suite [10]. In the

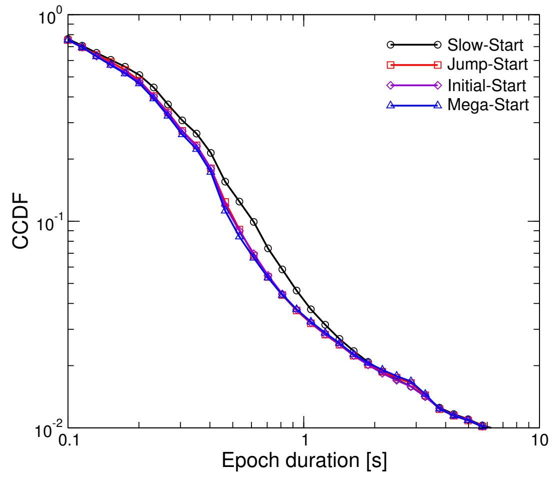


Figure 6.50: Distribution of the epoch time duration for end-to-end fast startups

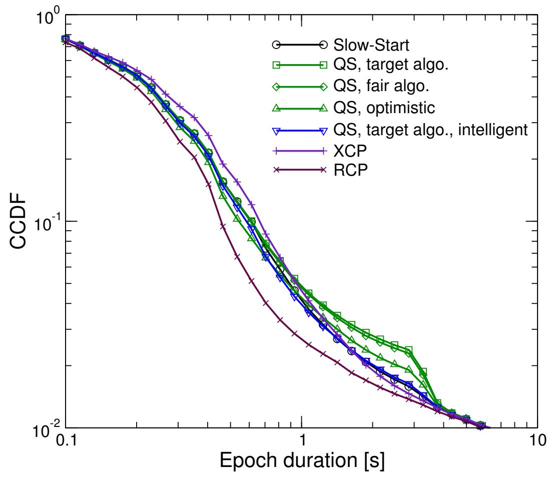


Figure 6.51: Distribution of the epoch time duration for network-supported fast startups

simulation setup, in total $N_{\text{hosts}} = 450$ client-server pairs are used, i. e., 50 per RTT value. The connection vectors are scheduled with an exponentially distributed IAT. The central bottleneck has a link capacity of $r_C = 10\,\text{Mbit/s}$ and uses a drop-tail buffer with $B = 50$. Since the traces represent an unknown mix of applications, it is impossible to measure application performance metrics in this setup. As a remedy, the epoch duration T_{epoch} is used as performance metric. Its definition is illustrated in Figure 3.13. For an epoch of the form $\mathbf{E}_i = (a_i, t_{a,i}, b_i, t_{b,i})$ it corresponds to the sum of the transfer times of the data (a_i and b_i) and the delay $t_{a,i}$.

6.4.5.2 Simulation results

Figures 6.50 and 6.51 show the resulting CCDFs of the epoch durations for the different flow startup schemes. In the presented experiment, the traces are scheduled so that the downlink load of the bottleneck is $\rho \approx 0.35$. Each presented measurement value corresponds to a simulated virtual time of about three hours. The first and foremost result of these simulations is that the differences between the different schemes are rather small. The reason is that the traces include many small data transfers that fit in today's initial CWND, and that the majority of the simulated path has an RTT smaller than 100 ms. Under these constraints the existing Slow-Start performs reasonably well for most transfers. Still, one can observe in Figure 6.50 that the epoch completion times that are of the order of several hundred milliseconds can be improved if the Jump-Start or Initial-Start scheme is used by all entities. A similar benefit also exists in case of Mega-Start, but the distribution tail also reveals an increased probability of delays of the order of 3 s, which is an indication for more Retransmission Timeouts.

Figure 6.51 presents the corresponding results for Quick-Start in combination with different router strategies. Each data transfer here issues a Quick-Start request of $q_{\text{req}} = 5.12\,\text{Mbit/s}$. In this usage scenario, Quick-Start is hardly of any benefit in most of the studied combinations. The reason is the fact that many requests are denied. As explained in Section 4.5.3.1, the Quick-Start mechanism can then have a performance worse than an unmodified TCP stack when many requests are denied. Figure 6.51 shows that there are only two combinations of Quick-Start algorithms that result in a slight improvement:

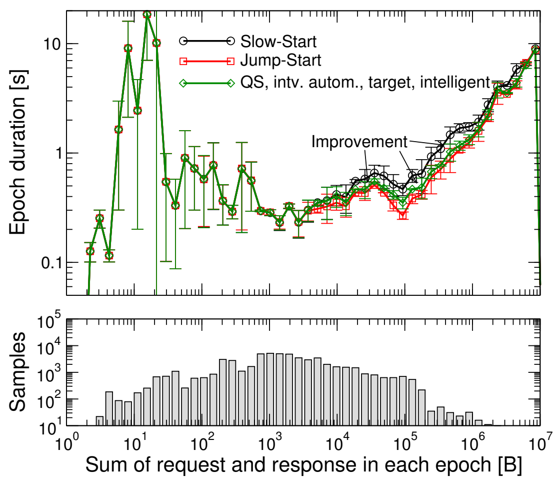


Figure 6.52: Breakdown of the performance depending on transfer sizes (load 6 %)

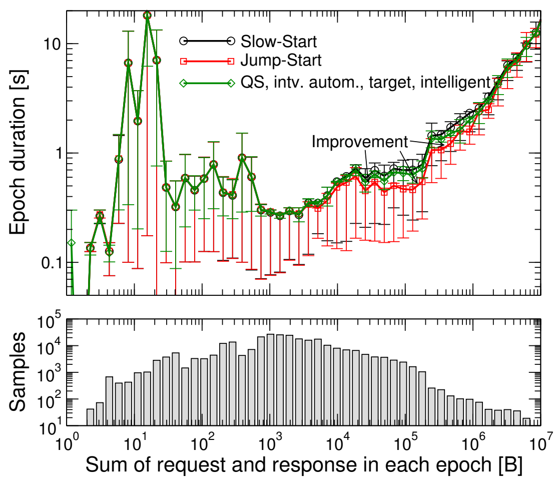


Figure 6.53: Breakdown of the performance depending on transfer sizes (load 35 %)

- *Approval control with the optimistic algorithm*: The oversubscription strategy proposed in Section 4.5.3.2 can deal with scenarios in which there are many Quick-Start request.
- *Intelligent activation*: Activating Quick-Start only for larger transfers ($\chi = 10\,\text{kB}$) avoids useless requests and thus uses the available bandwidth more efficiently.

Again, the performance of XCP is even worse than the one of a unmodified TCP stack. In contrast, RCP significantly outperforms any TCP-based scheme in this setup, since it speeds up the large number of rather short transfers.

The reason for the small total improvement can also be observed in Figure 6.52 and Figure 6.53, which depict the epoch durations as a function of the data size transported within that epoch, as well as the 25 % and 75 % quantiles. In Figure 6.52, the average downlink load is small, whereas the other diagram shows results for a moderately loaded link. As examples, the performance of Slow-Start, Jump-Start, and Quick-Start with intelligent activation are compared. Both figures show that for transfers with size $s > 10\,\text{kB}$ the average epoch duration is indeed reduced by up to some hundred milliseconds. For Jump-Start, the improvement is almost independent of the load. Quick-Start, in contrast, only improves the performance if the link load is small. The histogram in the lower part of both figures confirms that the majority of transfers is smaller than 10 kB and can thus hardly benefit from any fast startup scheme. In other words, only selected transfers can indeed benefit. Similar graphs for XCP and RCP are published in [181].

In order to provide a more systematic insight into the differences between the schemes, Figure 6.54 and Figure 6.55 study the trade-off between speedup and packet loss: The x-axis depicts the downlink packet loss probability, and the y-axis the 5 % quantile of the epoch duration for different downlink load conditions. The former metric characterizes the aggressiveness of an approach, whereas the quantile is one possibility to quantify the speedup of longer transfers. In the ideal case, a fast startup should reduce the epoch durations without causing additional packet losses. In this portfolio presentation, this ideal behavior would correspond to a vertical shift in down-direction.

With an unmodified stack, the packet loss probability is of the order of 1 % in the given scenario, and the epoch durations increase with the load, as to be expected. Figure 6.54 reveals that Jump-Start with its default parametrization ($K_{\text{data}} = 64\,\text{KiB}$) reduces the epoch duration but

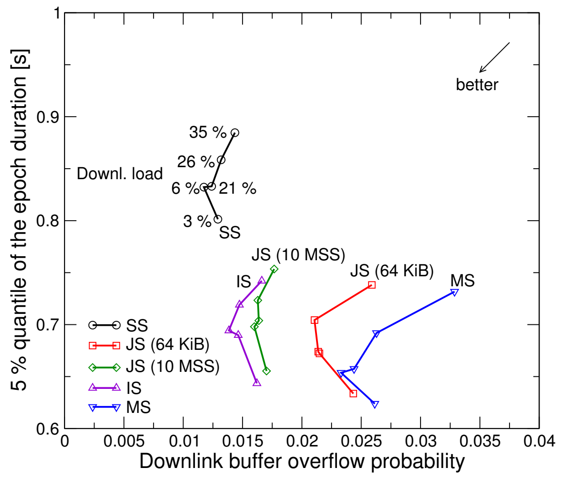


Figure 6.54: Speedup vs. packet loss for end-to-end schemes

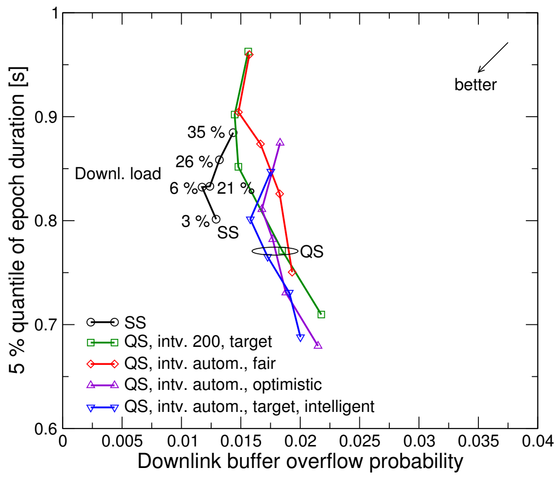


Figure 6.55: Speedup vs. packet loss for network-supported schemes

moderately increases the packet loss ratio by about 1 %. Initial-Start, which just increases the initial window to $w_{\text{iw}} = 10$, results in a similar speedup but less packet loss. In order to further compare this difference between Jump-Start and Initial-Start, simulations have also been performed with a comparable Jump-Start parametrization ($K_{\text{data}} = 10 \cdot L$). Then, both schemes have almost the same performance. One can conclude from this result that in a realistic traffic mix with many small transfers, there is only *few benefit of using rate-pacing* if the burst size is small. If Mega-Start is used by all transfers, the speedup is even larger, but this comes as the cost of rather large packet loss probabilities that can exceed 3 % if the load on the bottleneck increases.

Figure 6.55 contains packet loss statistics in the same setup for different Quick-Start usage scenarios and again relates them to the 5 % quantile of the epoch completion time. The diagram again confirms two findings:

1. *Quick-Start improves the performance only if the link load is rather small.* In this case, the speedup is comparable to end-to-end fast startup schemes. Quick-Start also moderately increases the packet loss probability compared to the Slow-Start mechanism.
2. On moderately loaded links, Quick-Start can result in longer transfer times if the endsystems frequently issue requests and if the target algorithm is used by the routers. This problem can be avoided by changing the algorithms either in the endsystems (*intelligent activation*) and/or in the routers (*optimistic algorithm*). Both variants have a similar benefit. An interesting result is that the optimistic algorithm does not result in significantly increased packet loss rates, even though bandwidth is oversubscribed. This outcome once again reveals that the oversubscription principle is superior to bandwidth pooling.

Of course, these results only represent a very small part of the overall parameter space. Similar experiments could be performed with many other setups. Still, the key differences between the different realization possibilities for fast startups would also be confirmed in other cases.

6.4.6 Summary of the performance experiments

The bottom line of these results is that the majority of data transfers in the current Internet would hardly benefit from a fast startup. Yet, for larger transfers there is an improvement. As

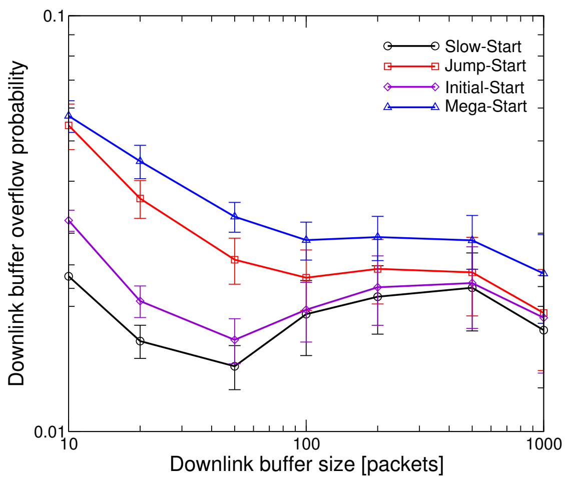


Figure 6.56: Influence of the buffer size on end-to-end schemes (load 35 %)

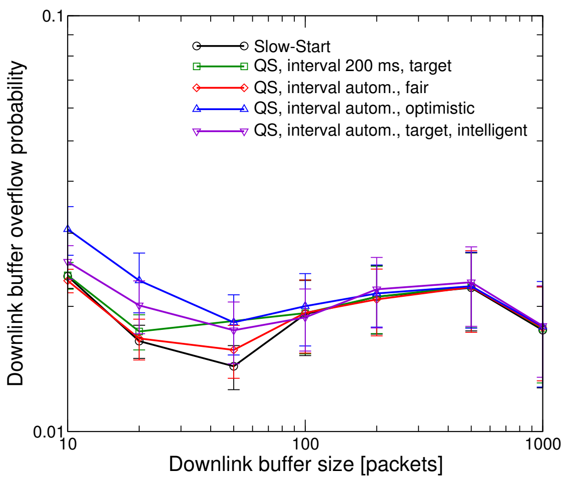


Figure 6.57: Influence of the buffer size on Quick-Start (load of 35 %)

to be expected, the end-to-end schemes Jump-Start and Initial-Start increase the packet loss probability, but only moderately. Quick-Start results in a similar performance benefit if it is intelligently used, but it causes less packet drops. The Mega-Start scheme is more aggressive and risky if it just stupidly starts with an approximation of the path capacity. It therefore cannot be used by all flows.

These results therefore also indicate that a *differentiated usage of fast startups* would make sense: Applications that are known to require broadband interactivity could explicitly activate the fast startup, while applications with less demands could just continue to use TCP's existing mechanisms, which are sufficient in many cases. This differentiation could be realized by the API proposed in Section 5.1.1.3. One possible realization is showcased in Section 7.1.

6.5 Robustness, fairness, and risk

6.5.1 Dealing with small buffers

Any fast startup congestion control schemes risks a higher packet loss rate. If the buffer in front of the bottleneck on the path is either large or uses AQM, it can accommodate the bursty traffic of a fast startup without or with only few packet drops. However, a small buffer will overflow very soon. As the drop of a packet wastes transmission resources on the path and requires additional processing both in the sender and the receiver, keeping the packet loss probability to a low value is an important design goal for congestion control mechanisms (cf. Section 4.1.2.1).

In order to study the impact of the size of the bottleneck buffer, the experiments with the trace-based workload have been repeated with different buffer configurations. All other parameters are set like in the previous section. Figure 6.56 and Figure 6.57 depict the packet drop probability in the bottleneck. In the given scenario with an average downlink load of $\rho = 35\,\%$, TCP's default congestion control causes a packet loss probability between 1 % and 2 %, which is a typical value for reasonably used Internet paths.

The buffer overflow probability has a complex dependency on the buffer size: In the given scenario, a buffer of the order of 50 packets minimizes the packet losses. If the buffer is smaller, it cannot accommodate bursts. If it is larger, it can buffer many segments. But this also means that the TCP senders aggressively increase their Congestion Window in Slow-Start and enter the Congestion Avoidance phase rather late, i. e., they risk overshooting. A very similar effect can also occur when link load is varied, i. e., the overall buffer overflow probability of a moderately loaded link can be smaller than the one of a lightly loaded link.

Figure 6.56 confirms that all end-to-end fast startup schemes increase the packet loss probability. The Jump-Start scheme increases the packet drop rate by a factor of about two compared to Slow-Start, if the buffer size is $B < 100$. If the buffer in front of the bottleneck is larger, there is hardly any difference to the Slow-Start. Interestingly, just increasing the initial window without rate pacing (Initial-Start) results in less lost packets and works well even if the buffer size is rather small. If the Mega-Start scheme starts with a rate equal to the full path capacity, it is significantly more aggressive, and it can result in packet drop probabilities of 5% or more. These numbers are consistent with the simulation results shown in Figure 6.54.

The key difference between end-to-end fast startup mechanisms and Quick-Start is that Quick-Start only enables the fast startup if there is available bandwidth on the path. Otherwise, it falls back to the Slow-Start. This means that the risk of packet loss is smaller. This expected result is confirmed by Figure 6.57. In general, all different Quick-Start usage scenarios have a packet loss rate that is of the same order of magnitude like the one of a scenario with default TCP stacks. There is slighty more packet loss in the more aggressive Quick-Start usage variants (*intelligent activation* in endsystems and/or *optimistic algorithm* in routers), but only if the buffer is rather small ($B < 100$), and the packet loss rate increases by less than 1 %.

In summary, these results illustrate the *fundamental trade-off* between potential speedup on the one hand, and a higher risk of packet loss on the other hand. Quick-Start is more conservative and works well even if the bottleneck buffers on the path have a very small size only, while end-to-end schemes result in an increased packet loss rate. In general, the buffer sizes of real network components vary over several orders of magnitude. This is why it is impossible to precisely forecast the impact of a fast startup scheme in a large network. Still, the results presented in this section indicate that fast startup schemes such as Jump-Start, Initial-Start, or Quick-Start do only moderately increase the buffer overflow probabilities, and operate quite well if the network components can accommodate bursts of some dozens of packets, which is known to be true in many cases [234].

6.5.2 Fairness compared to TCP's default congestion control

Another implication of most new fast startup congestion control schemes is that they may affect competing connections that use TCP's default congestion control, unless the traffic is completely isolated. In order to study the negative impact of aggressive fast startup schemes on a standard-compliant TCP flow start, a simulation scenario has been set up where only half of the senders enable a fast startup scheme, while the rest uses an unmodified stack. Exactly like in the previous simulation setups, there are $N_{\text{hosts}} = 450$ clients and servers with nine different RTTs, i. e., 225 pairs use Slow-Start and 225 a fast startup scheme. In order to ensure that in average both classes of endsystems transport workload with the same statistical properties, the workload is here given by a synthetic traffic model with response sizes according to a truncated pareto distribution ($m = 14\,\text{kB}$, $\alpha_{\text{pareto}} = 1.1$, $K_{\text{pareto}} = 10\,\text{MB}$), which matches the tail characteristics of the traffic traces (cf. Section 3.3.3.2). This workload model allows to simulate higher link utilizations without requiring a very large number of endsystems.

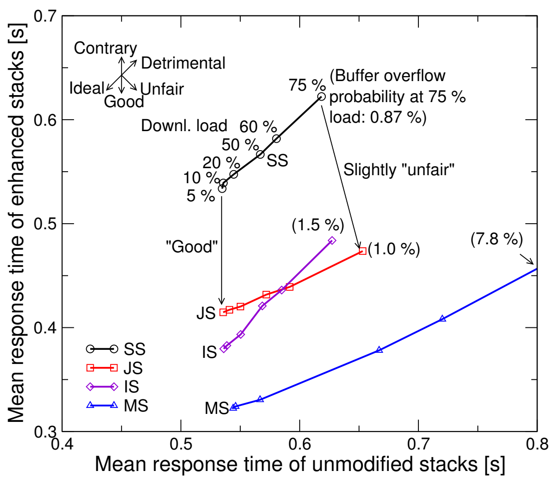


Figure 6.58: Mixed usage of default and end-to-end fast startup congestion control

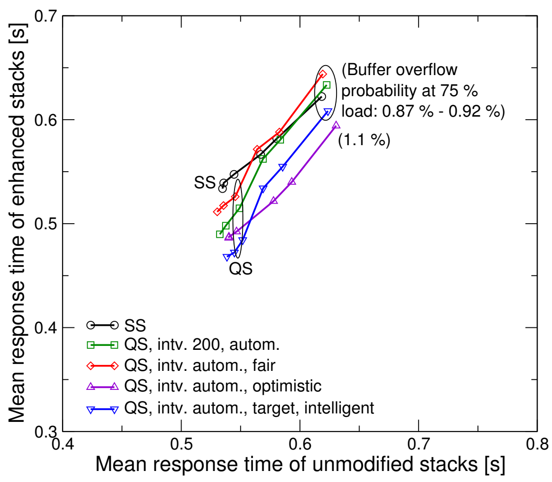


Figure 6.59: Mixed usage of default and Quick-Start congestion control

There are five possible outcomes of the interaction between a default and an enhanced congestion control [205]:

- *Ideal*: Both schemes experience better performance. This is a rather unlikely effect.
- *Good*: Using the new schemes results in a better performance, but the performance for other ones is not worsened.
- *Unfair*: There is a speedup at cost of endsystems that use the default congestion control.
- *Contrary*: The enhanced stacks are slower than default ones.
- *Detrimental*: The performance is worse in both cases.

In Figure 6.58 and Figure 6.59, the x-axis and y-axis represent the response times measured for the two groups, for mean downlink utilizations up to 75 %. In this representation, an *ideal* or *good* result is indicated by a shift to the lower-left or lower part of the diagram, respectively. According to Figure 6.58, both Jump-Start and Initial-Start are rather fair, i. e., they speed up their own transfers without significantly slowing down connections that use the default TCP congestion control. The Mega-Start scheme with the assumed parametrization is unfair, which is consistent to the other simulation results in the previous section.

The corresponding Quick-Start results are depicted in Figure 6.59. As to be expected, Quick-Start hardly degrades the performance of connections that use a default congestion control, but its speedup compared to Slow-Start is also smaller.

To sum up, these experiments show that using a fast startup may not necessarily result in severe unfairness problems for connections that do not use them, even if a more aggressive behavior inherently results in such a risk. Of course, it can be reduced by additional signaling along the path. A further option to mitigate this problem would be an identification and isolation of the traffic that uses a fast startup congestion control, e. g., by different DiffServ classes, but such a scheme is not further studied in this work. There is also a trade-off between unfairness and convergence speed: The experiments in Section 6.3.2 have shown that a very aggressive end-to-end flow startup improves the time until a bottleneck is equally shared by several connections. Without network support, the convergence speed of end-to-end loss-based congestion control algorithms can only be improved by causing some packet loss to the competing traffic.

Table 6.3: Consequences of false capacity assumptions in network components

Scheme	Capacity underestimation	Capacity overestimation
Quick-Start	- Less bandwidth granted to requests - Slower flow startup or fallback to SS - Target use case of Quick-Start	- Approved bandwidth too large - Transient packet loss due to overshooting
XCP and RCP	- Link permanently not fully utilized - Empty queues	- Extremely long persistent queues, or - Permanently large packet loss rate if the buffer size is small

6.5.3 Robustness against imprecise information

6.5.3.1 Challenge of capacity estimation

A fundamental requirement for any congestion control scheme is *robustness*, i. e., the ability to deal with unpredictable variations of the environment and erroneous input data. Of particular importance for network-supported congestion control schemes is the correctness of information about link characteristics. As already mentioned in Section 4.7.1.2, routers cannot know the available bandwidth on a link in all cases. Still, the control equations of network-controlled schemes such as XCP and RCP assume that the link capacity c is exactly known. The impact of wrong assumptions about link characteristics is hardly addressed in published work on network-supported congestion control, apart from some efforts to extend XCP to networks with shared access [1]. In the following it is shown that link capacity estimation errors are a severe problem for XCP and RCP. This is a key advantage of Quick-Start, which is much more robust.

Let r be the true capacity of an outgoing link, and let c be the capacity assumed by a network component supporting a network-supported congestion control scheme. Then, two different link capacity estimation errors can occur:

- *Capacity underestimation*: $c < r$
- *Capacity overestimation*: $c > r$

The impact of these two different cases on Quick-Start, XCP, and RCP is compared in Table 6.3 and analyzed in the following subsections.

6.5.3.2 Robustness of Quick-Start

Quick-Start is explicitly designed to operate with a *conservative estimation* of the true link capacity. For this purpose, the target admission control algorithm includes the parameter θ, which shall ensure that only a certain part θ of the link capacity is granted to Quick-Start requests. Setting $\theta < 1$ reduces the efficiency, since less requests can get approved, but the TCP congestion control continues to work well. And there is still a certain speedup if only a certain part of the link capacity is available for Quick-Start requests [199].

This effect is confirmed in Figure 6.60, which shows the traces of a long-lived flow for three different cases ($\tau = 200\,\text{ms}$): Correct knowledge or the link capacity ($c = r = 10\,\text{Mbit/s}$), underestimation ($c = r/10 = 1\,\text{Mbit/s}$), and overestimation ($c = 10 \cdot r = 100\,\text{Mbit/s}$). In the case of underestimation, only a small rate is granted, i. e., the flow startup is just slower. If the approval control of a QS-enabled router assumes a too large capacity $c > r$, the granted rates may be too large, and the sender then temporarily sends with a too large data rate. As shown in Figure 6.60,

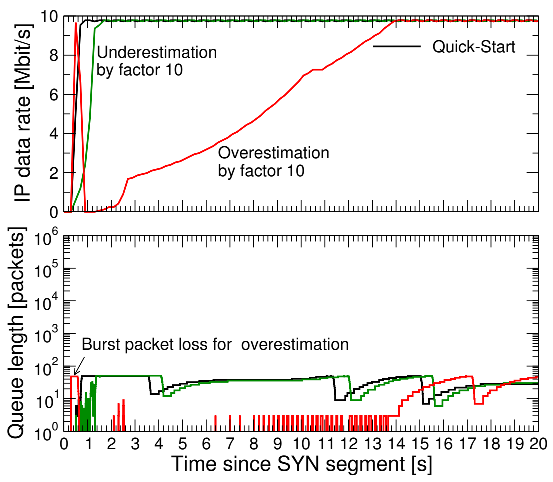


Figure 6.60: Impact of capacity estimation errors on Quick-Start

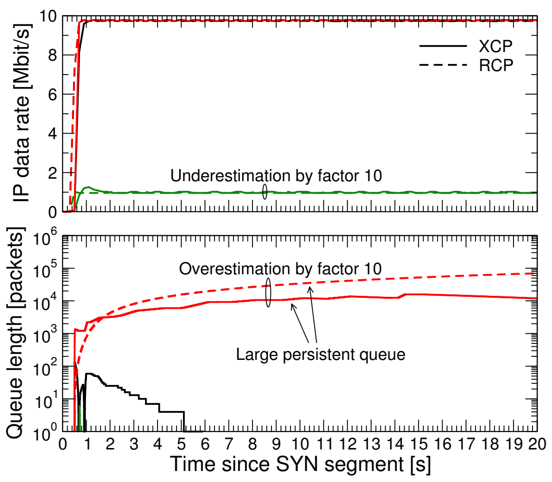


Figure 6.61: Impact of capacity estimation errors on XCP and RCP

this results in an overshooting and multiple lost packets. TCP then suffers from a Retransmission Timeout, but it detects and solves the problem within few RTTs. In the presented scenario, the sender requires several seconds until it can fully utilize the link. But such an overshooting could also occur in normal TCP operation. Furthermore, the problem only exists if the requested rates are in total indeed larger than the true link capacity, i. e., there is no harm if there are small Quick-Start requests only.

The risk of overshooting can be significantly reduced by setting the approval target $\theta \cdot c$ to a worst-case assumption of the available bandwidth, i. e., $\theta < 1$. For instance, on a router interface towards a shared Ethernet network, $\theta \cdot c$ could be set to 10 Mbit/s only, which is the minimum Ethernet link capacity. This conservative setting would work reasonably well even if all connected Ethernet devices support a higher data rate (100 Mbit/s, 1 Gbit/s, . . .).

6.5.3.3 Robustness of XCP and RCP

XCP and RCP are much more *susceptible to erroneous link capacity information*. If the link capacity is underestimated ($c < r$), they cannot efficiently utilize the link: Since Equation (4.20) and Equation (4.21) provide no positive feedback once the assumed capacity is reached, the maximum possible utilization of the link is $\rho = c/r$. For instance, in Figure 6.61, the achieved peak throughput is only 10 % if the link capacity is underestimated by factor $r/c = 10$. Even if several connections shared the link, the efficiency would not be improved. This is the drawback of network-controlled congestion control. It is in theory possible to identify such underutilization by an additional control loop [1], but only if the traffic conditions are rather stable.

In fact, both XCP and RCP can only efficiently utilize a link if the assumed capacity c is *at least* as large as the true capacity r. However, both schemes are also not very robust against capacity overestimation neither: If $c > r$, the actual link usage is $x(t) = c$, and all excess traffic is queued. As an increase of the queue length results in negative rate feedback, the system should remain stable. However, it can easily be shown that a stable state is not necessarily reached. In the following, the stability bound for RCP is derived. To the best of the knowledge of the author, this condition has not been published elsewhere. XCP suffers from exactly the same problem, but an analysis is more difficult due to the complex control law.

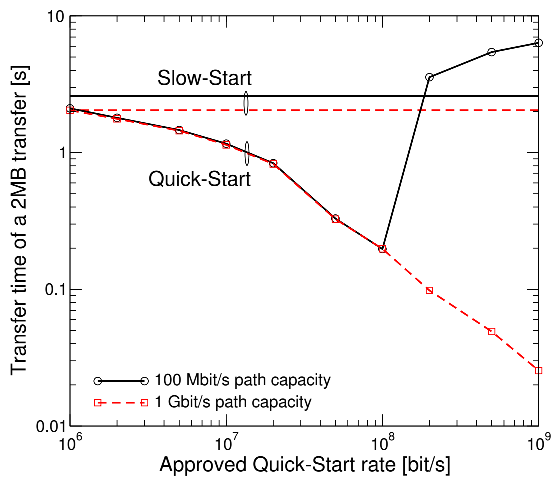


Figure 6.62: Impact of Quick-Start capacity overgranting over the experimental path

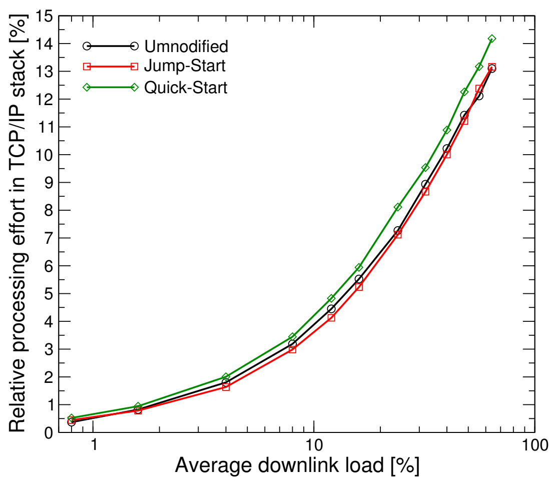


Figure 6.63: Server-side processing effort in the Linux stack obtained by kernel profiling

In steady state, the feedback of RCP according to Equation (4.21) must be zero:

$$\alpha_{\text{RCP}}\left(\theta \cdot c - r\right) - \beta_{\text{RCP}} \frac{b}{d_{\text{avg}}} = 0. \tag{6.13}$$

Without loss of generality, θ is here assumed to be 1. The variable b refers to the queue length. Assuming a sufficiently large buffer, the average RTT d_{avg} is increased by the time that is required to pass through queue: $d_{\text{avg}} = \tau + \frac{b}{r}$. By substituting d_{avg} in Equation (6.13) one can determine the persistent queue length as

$$b = \frac{(c-r)\cdot\tau}{1 + \beta_{\text{RCP}}/\alpha_{\text{RCP}} - c/r}. \tag{6.14}$$

As the queue length must be finite, the denominator must be larger than zero. From this follows a stability condition:

$$\frac{c}{r} < 1 + \frac{\beta_{\text{RCP}}}{\alpha_{\text{RCP}}}. \tag{6.15}$$

The stability of RCP in case of capacity overestimation thus depends on the parameters α_{RCP} and β_{RCP}. For the commonly suggested parameter set $\alpha_{\text{RCP}} = 0.4$ and $\beta_{\text{RCP}} = 0.226$, the stability condition is $c/r < 1.565$. In other words, the capacity must be known with a maximum error of 50 %. Other suggested parameter sets such as $\alpha_{\text{RCP}} = 0.1$ and $\beta_{\text{RCP}} = 0.1$ allow an estimation error by about factor three. If the estimation error is larger, the queue length will grow without restraint.

This effect can be observed in Figure 6.61, both for XCP and for RCP. As the condition of Equation (6.15) is not fulfilled in this simulation setup, the number of queued segments increases approximately linear in time. In practice, this means that packet loss cannot be avoided, and that XCP and RCP must fall back to another operation mode (e. g., some TCP-like algorithms), which has not been specified so far and renders the whole approach somehow useless.

6.5.3.4 Summary

In summary, the Quick-Start mechanism works rather well if the order of magnitude of the available bandwidth is known, and the penalty of configuring the approval control to a too small target rate is rather small. If the approval control is configured with a too large target capacity, transient packet loss occurs, but the overall impact is rather small. This is also confirmed by measurements on the real network path shown in Figure 6.62: If the approved Quick-Start rate is larger than the true path capacity, the transfer times are typically larger than the ones in Slow-Start, as packet loss occurs, but the performance difference is rather small.

In contrast, the network-controlled schemes XCP and RCP are not very robust against capacity misestimation. If the assumed link capacity is too small, the link cannot be efficiently utilized. If it is significantly overestimated, it is possible that the control loops get unstable. Without modifications, XCP and RCP can only be used if the available bandwidth on links is known by approximately a factor of two. This condition is not necessarily fulfilled in real environments and thus a fundamental limitation.

6.6 Complexity and costs of network support

6.6.1 Computational overhead of endsystem functions

All fast startup congestion control schemes require modifications that possibly result in additional processing effort in the endsystems and – if they are network-supported – also in network components. There are only few published studies on the computational overhead caused by such new mechanisms. The availability of comparable implementations in the same network stack enables a fair comparison of this overhead.

An additional function required by most fast startup mechanisms is fine-grained rate pacing, which requires timers and increases the complexity of the network stack. In general, it is difficult to quantify the impact of such new protocol mechanisms, since metrics such as CPU usage are very specific for an implementation and a given system. Also, the overhead of new mechanisms can only be measured if they can be totally isolated from existing functions, which is difficult in a complex protocol stack. Still, experiments can provide some insight into the processing overhead of new fast startup congestion control schemes.

Figure 6.63 shows the results of a server-side profiling analysis of the measurement setup that corresponds to Figure 6.44. The presented numbers refer to the CPU usage of all IP and TCP function calls compared to the total processing effort in the system within a measurement duration of 30 s. The numbers have been obtained by an “oprofile” profiler that instruments the Linux kernel. As to be expected, the relative processing effort in the TCP/IP stack increases with the network load. The higher the load, the more packets have to be processed. One can also observe that the additional processing overhead of an end-to-end scheme such as Jump-Start is negligible. In some samples, the measured system load is even slightly smaller than the one of an unmodified TCP/IP stack, which is probably caused by the shorter transfer durations.

It is important to note that the workload scenario challenges the rate pacing implementation because many data transfers are completely realized during the rate pacing phase. This small overhead indicates that the usage of *rate pacing does not cause significant performance problems*. End-to-end schemes other than Jump-Start have similar characteristics. All in all, the additional processing by a fast startup is simple compared to the complexity of the Linux TCP/IP stack as a whole, and the new mechanisms are only active for rather short periods of time. This observation is consistent with other studies [94, 102] that also found a small overhead of rate pacing with a millisecond timer resolution.

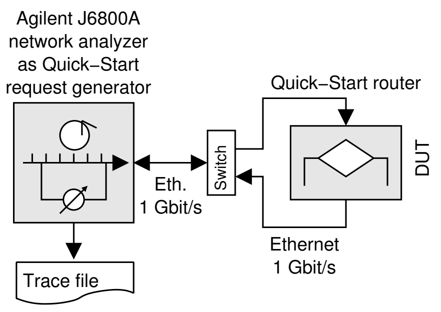


Figure 6.64: Setup of the load test for Quick-Start routers

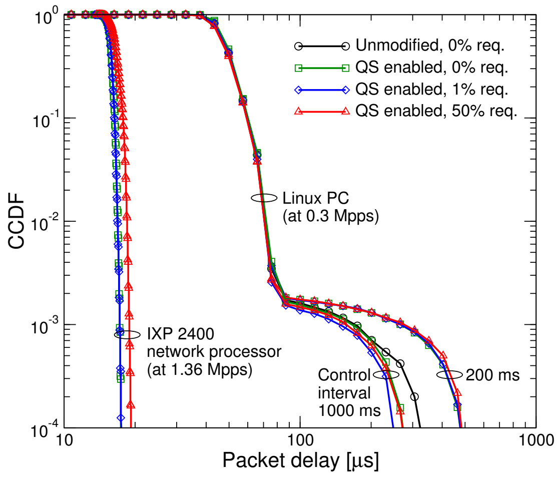


Figure 6.65: Delay distribution of packets through a router under load

Unlike end-to-end schemes, Quick-Start TCP causes some measurable overhead in the sender. It is of the order of 1 % of the total CPU load and increases when there is more traffic. The profiling results reveal that the most significant contribution is the additional traffic metering in the IP layer, which counts every packet. On the one hand, this processing overhead in the endsystem shows that network-supported congestion control indeed comes at some cost. But on the other hand, the overhead is rather small, i. e., Quick-Start is still an uncomplicated signaling mechanism. The overhead could be reduced by optimized implementations. For instance, traffic metering could be performed more efficiently in the network interface card itself, e. g., by packet counters.

6.6.2 Computational overhead of router functions

The key challenge of router-supported congestion control is the additional packet processing that is required in network components. This section substantiates that the computational overhead of Quick-Start in routers is small. These measurement results are partly published together with Hauger [204, 203, 84].

The computational overhead of processing the Quick-Start IP options is studied both in the Linux kernel and in the high-speed network processor implementation. Both Quick-Start implementations are introduced in Section 5.3. These results have been obtained from *stress tests* that challenge a router by a large *packet rate*, which is measured in *Packets Per Second* (PPS). The load test setup is depicted in Figure 6.64. In order to ensure that the bottleneck is indeed the router processing speed and not the link capacity, the used packets have a length of 50 B only. A network analyzer is used both for the load generation and for the measurement of the delay with high-precision timestamps. Due to limitations of the network analyzer, delay measurements can only be performed for 700,000 subsequent packets. In the experiment, either 0 %, 1 %, or 50 % of the sent packets include an IP option with a Quick-Start request. A share of 1 % is a realistic assumption if the Quick-Start mechanism will become widely supported. A much larger ratio should never occur when it is used as foreseen. The measurements with a ratio of 50 % were performed in order to study the impact of a potential DoS attack.

Table 6.4: Quick-Start implementation performance characteristics (adapted from [203])

Network stack used in router	**Share of QS req.**	**Linux personal computer ... Peak rate**	**Mean delay at 0.3 Mpps**	**IP forw. effort at 0.3 Mpps**	**Network processor ... Peak rate**	**Mean delay at 1.36 Mpps**
Unmodified	0 %	0.33 Mpps	50 μs	2.5 %	3.9 Mpps	16 μs
Quick-Start	0 %	0.33 Mpps	50 μs	2.6 %	3.9 Mpps	16 μs
Quick-Start	1 %	0.33 Mpps	50 μs	2.8 %	3.9 Mpps	16 μs
Quick-Start	50 %	0.33 Mpps	50 μs	8.1 %	3.8 Mpps	17 μs

Figure 6.65 prints the complementary cumulative distribution function of the delay that packets experience when traversing a router. In general, the delay through the Linux PC is significantly larger than the one through the router realized with the network processor. In both cases there is hardly any difference between the measurements with and without Quick-Start options. This shows that *Quick-Start option processing causes only very small additional delays*. In case of the Linux implementation, the traffic estimation interval Δ does have an influence. If Δ is short, i. e., less than one second, the traffic statistics are more frequently updated, and this effect can be seen in the tail of the distribution function. But its impact on the average delay is negligible.

Table 6.4 gives some further insight into the impact of Quick-Start on the router performance. Again, the effort of processing additional IP header fields in the fast path is found to be not very high. Concerning the peak throughput, i. e., the maximum packet rate that can be transported without packet loss, the Quick-Start implementations do not suffer from a significant degradation compared to a corresponding router without Quick-Start support. The maximum packet rate of the network processor is about 3.9 Mpps without and with Quick-Start support. With full-sized IP packets, this would correspond to a throughput of multiple Gbit/s. For very frequent Quick-Start requests in every second packet there is a small decrease. As described in [84], the synchronization among fast path processing entities can become a bottleneck if a very large number of options has to be processed in parallel and if the synchronization of state information is required, e. g., because of the usage of the target algorithm. However, a ratio of 50 % should never occur in normal usage. With a reasonable traffic mix with 1 % QS request, there is no difference between a default router and a QS-enabled router, independent of the used synchronization mechanism. The synchronization can also be completely avoided if no global state variables are modified on a per-packet basis, i. e., the Quick-Start approval control with the *optimistic algorithm*.

On a PC-based router, the maximum achieved packet rate is about one order of magnitude smaller and of the order of 0.33 Mpps. Again, the Quick-Start option processing hardly affects the peak packet rate and the delay. In the PC used in this experiment, the bottleneck is likely to be the internal bus between CPU and network interface card. Interestingly, in some experiments the peak packet rate was even some percent larger when many packets carry a Quick-Start option, which is probably a kernel-internal side effect.

A kernel profiling analysis was used as well in order to detail these results: Table 6.4 lists the CPU load share of all method calls that handle the packet forwarding and process IP options, both for the unmodified and the patched kernel. Without any Quick-Start options being present, there is a measurable overhead, but it is small. The additional effort is mainly caused by the traffic metering that counts every IP packet. As already mentioned, it could be avoided if the link utilization was already available in the network interface card. If 1 % of the packets include a Quick-Start option, the computational effort of the IP forwarding is hardly affected. Only if the share of Quick-Start packets is very high, there is a measurable impact, which results from the general processing of IP options and the Quick-Start algorithms. A detailed analysis shows that the overhead caused by Quick-Start methods is about 4 % of the total CPU load at a packet rate of 0.3 Mpps, which is close to the peak rate. It is mainly caused by the functions that modify the Quick-Start IP option (modification of the rate, recalculation of the random nonce, etc.).

One main advantage of Quick-Start compared to network-controlled congestion control such as XCP and RCP is that the additional packet processing is only required for few packets in realistic usage scenarios. According to measurements published in [226, 84], the prototype network processor implementation of XCP achieves a maximum throughput of about 1.4 Mpps only, and *all* packets experience an additional internal delay of the order of 10 μs due to the required synchronization of global variables. And this problem gets worse as parallelism increases [226, 84]. These results reveal that it may be difficult to implement XCP on router architectures with a high degree of parallel packet processing. A comparable Quick-Start implementation does not suffer from these problems.

By design, RCP does not require a per-packet synchronization. References [58] and [102] both report a very small computational overhead of an unpublished RCP implementation in the Linux networking kernel. According to Dukkipati [58], the RCP-related processing is only 2.6 % of the total processing for IP packet forwarding in the Linux kernel. This statement provides some evidence that Quick-Start and RCP have probably a similar, moderate realization complexity in network components, i. e., in terms of implementation complexity one should be able to implement the required additional functions in high-speed routers.

7 Applicability case studies

This chapter presents two short case studies that illustrate the real-world applicability of the fast startup congestion control mechanisms. They also provide evidence that the performance of novel network-challenging applications can be improved perceptively. These results complement the synthetic experiments in the previous chapter. The first use case proposes a simple HTTP protocol extension that would allow Web applications to comply with given performance requirements by the corresponding activation of a fast startup scheme. This mechanism could be part of a simple *Web performance requirement signaling architecture*, in which the network stack takes into account application-level SLA specifications for response times as introduced in Section 3.1.3.3. The usefulness of this architecture is demonstrated with a case study that implements these extensions both in a Web browser and a Web server. Exemplary performance results are provided for the *OpenStreetMap* (OSM) Web page, which is a typical example for a Website where larger images have to be retrieved. The second considered example are applications dealing with *three-dimensional* (3D) content. Emerging 3D Web applications are a typical representative of a *broadband interactive application*, since interactivity is a key usability objective. This case study substantiates the claim that "mid-sized" transfer sizes are common in such emerging applications. The hypothesis is backed by experiments that combine a prototypical 3D visualization application with fast startup congestion control schemes.

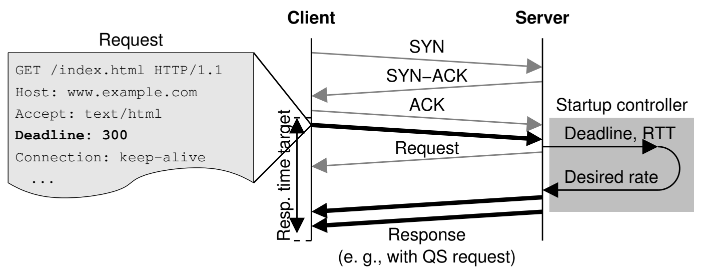
This short chapter only provides examples and proof-of-concepts for the interaction of real-world applications and fast startup congestion control. In this respect, all presented results are illustrative, and many application-specific aspects are not investigated comprehensively. The two presented use cases could also be starting points for future work beyond this thesis.

7.1 A new Web performance requirement signaling architecture

7.1.1 Congestion control with response time deadlines

As explained in Section 3.2.1.2, there is no established solution to map application-level performance requirements to the TCP/IP network stack. Fast startup congestion control could be that missing piece in the Internet architecture, provided that the startup parameters can be explicitly configured by applications, e. g., by socket options as proposed in Section 5.1.1. Obvious use cases are applications that have a desired *maximal response time*, which could be mandated by the delay tolerance of human beings, or by SLAs as shown in Figure 3.4. Fast startup congestion control can be a mechanism to fulfill such SLAs, provided that the required bandwidth is indeed available on the path.

In the simplest case, a response time target is known by the sending application, which can then activate a fast startup scheme accordingly. However, in case of a client-server Web application with asymmetric communication characteristics, the server must decide whether and how to



```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: text/html
Deadline: 300
Connection: keep-alive
  ...
```

Figure 7.1: HTTP-based signaling of response time requirements; in the presented example a non-standardized HTTP-extension sets a deadline of 300 ms

activate a fast startup. Even though the server is typically able to determine the size of the requested content, it can not necessarily make an informed choice how to send the data, since it may not be aware of the urgency of the data transfer. The latter depends on current status of the client application and the user preferences, which may, if at all, only be known in the client. In such cases, it makes sense to explicitly signal the performance requirements from the client to the server. As illustrated in Figure 7.1, a client could inform a server about a response time deadline by a simple HTTP extension. This information could then be used by the server to configure a fast startup.

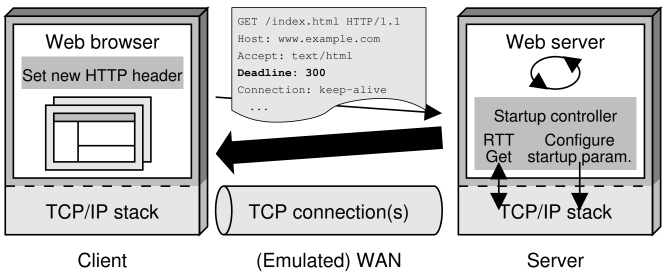
Fast startup schemes with an explicit application interface, such as Quick-Start and Mega-Start, can be configured to start with a certain sending rate. Then, an application function is required that decides which rate to use. If a *response time target* T_{target} is either locally known or signaled from the other endsystem like in Figure 7.1, the application can select an initial rate Q_{req} as follows:

$$Q_{\text{req}} = \frac{s}{T_{\text{target}} - d}. \tag{7.1}$$

This rule assumes that the application knows the amount of data s to be sent at the beginning of the transfer, which is typically true for downloadable content. Additionally, the estimated RTT d must be obtained from the TCP/IP network stack. In case of Linux, this is easily possible with help of a socket option that provides access to TCP state information.

One could also determine a reasonable initial rate by rules other than Equation (7.1). For instance, it has been suggested that an application could request for a Quick-Start rate that is sufficient to transmit a given amount of data in a single RTT [192, 191]. This approach would be very similar to the Jump-Start design philosophy. In combination with Quick-Start, Equation (7.1) could also be modified in order to take into account the one RTT that is required for signaling. Alternatively, the RTT could not be taken into account at all ($Q_{\text{req}} = s/T_{\text{target}}$), since it cannot be controlled. These straightforward variants and extensions are left for further study. Due to the complex communication patterns of modern Web applications, the exact value of the rate is less important and setting it to the right order of magnitude will suffice.

Determining a reasonable value for T_{target} per individual request is a non-trivial issue. For instance, if a Web transaction consists of multiple requests, it is required to subdivide an overall delay target into individual deadlines per server and per connection. The optimal solution depends on the structure of the content. Yet, since a congestion control scheme anyway provides no guarantee that the target delivery times are indeed met, one can expect that simple approximations are sufficient. Also, the penalty of choosing a value that is too large or too small is rather low; in both cases, the presented fast startup TCP extensions will not perform much worse than the standard algorithms and deliver data more or less as fast as an unmodified stack would



```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: text/html
Deadline: 300
Connection: keep-alive
...
```

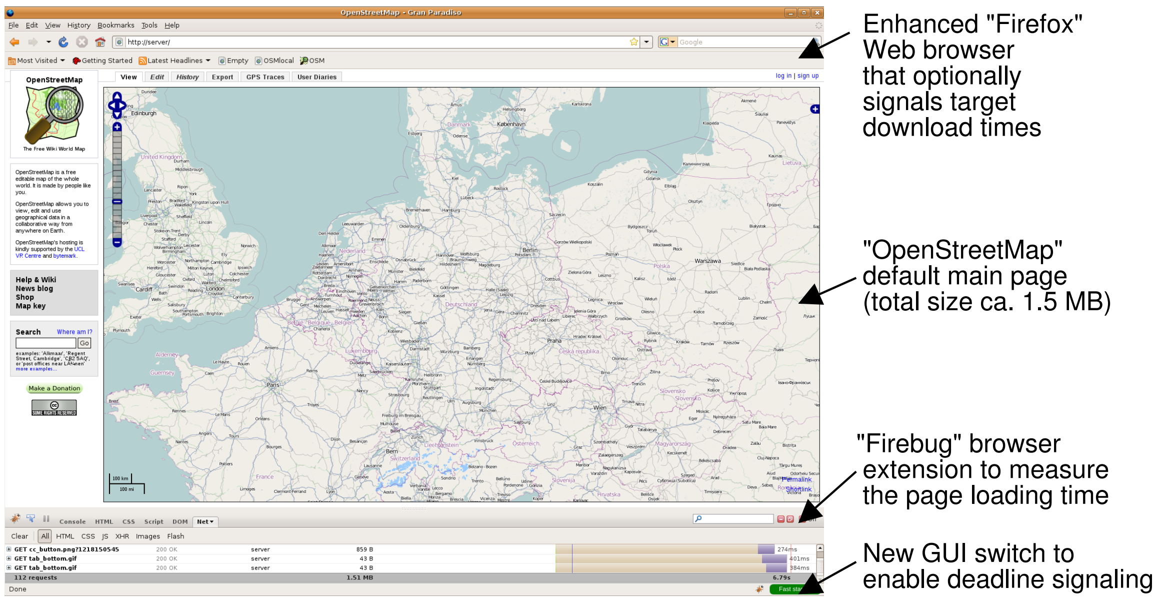
Figure 7.2: Web application with deadline compliance extensions

do. Obviously, fast startup schemes without explicit application interface (i. e., Jump-Start, Initial-Start) do not offer such a fine-grained control, but they still will transport the content faster if they are enabled.

7.1.2 Proof-of-concept realization

It is simple to implement the suggested application-level signaling of response time targets and to enhance Web application by the corresponding API to the fast startup congestion control. Figure 7.2 shows the modifications that are required in a client and server that communicate by HTTP. In the client, only two changes are required: It must determine an appropriate response time target, e. g., by user configuration, and it must signal this value by an additional HTTP header line. The server must have a corresponding parser and it must store the received value. Once the corresponding content is available, it must determine the parameters, i. e., the content size and the RTT, and then activate the fast startup accordingly by the sockets interface.

In order to prove that these extensions are very lightweight, both the "Firefox" Web browser as well as the "lighttpd" Web server have been enhanced to support the signaling of target response times. A screenshot of the enhanced "Firefox" browser is printed in Figure 7.3. In


Enhanced "Firefox" Web browser that optionally signals target download times
"OpenStreetMap" default main page (total size ca. 1.5 MB)
"Firebug" browser extension to measure the page loading time
New GUI switch to enable deadline signaling
OpenStreetMap - Gran Paradiso
Console HTML CSS Script DOM Net
Clear All HTML CSS JS XHR Images Flash
Done
Fast sta

```
GET cc_button.png?1218150545   200 OK   server   859 B    274ms
GET tab_bottom.gif             200 OK   server    43 B    401ms
GET tab_bottom.gif             200 OK   server    43 B    384ms
112 requests                                     1.51 MB   6.79s
```

Figure 7.3: Screenshot of the enhanced "Firefox" Web browser showing the "OpenStreetMap" main page and an integrated measurement of the page loading time

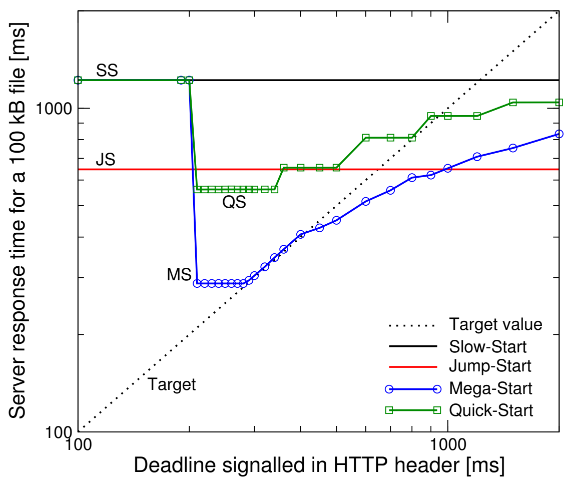


Figure 7.4: Compliance to a signaled response time deadline for a single object

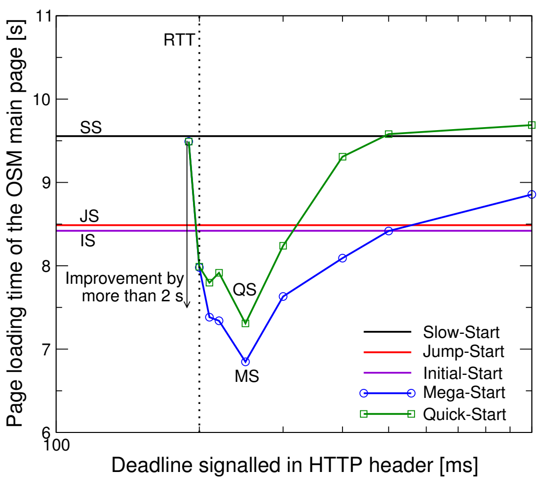


Figure 7.5: Influence of the deadline on the "OpenStreetMap" Web page loading time

this proof-of-concept realization, the response time target is statically configured in the browser configuration. The performance signaling can be activated or deactivated by a new GUI switch element that is shown in the status line. The addition of the new HTTP header only requires about 20 additional lines of code in the "Firefox" browser, and the GUI extension is straightforward, too. In the used Web server, about 70 lines of code are needed in order to parse the HTTP header and activate the fast startup. These small numbers confirm that such an extension is possible with almost negligible effort.

7.1.3 Exemplary measurement results

In order to illustrate the fundamental impact of the response time target signaling, Figure 7.4 shows a simple test result with a single object. In the experiment, a 100 kB file has been retrieved from the modified Web server over a link with capacity $r = 10$ Mbit/s and an emulated RTT of $\tau = 200$ ms. The target time is varied between 100 ms and 2 s. In Figure 7.4, one can clearly distinguish the startup schemes that support a parametrization by the application (Quick-Start and Mega-Start) from those that do not have an application interface. In the latter case, the performance is independent of the application's need and may, or may not, meet the required deadline. In case of Mega-Start and Quick-Start, a smaller deadline improves the server response time. Mega-Start meets the response time target as long as this is possible over a 10 Mbit/s link. If the delay target is impossible because it is smaller than the RTT, the Web server is programmed not to activate the fast startup. This effect can be observed in Figure 7.4 for $T_{\text{target}} < \tau$. Quick-Start shows a similar behavior, but it does not meet the signaled deadline. The initial rate determined by Equation (7.1) is not large enough if Quick-Start is used with late-activation, i. e., if there is a signaling delay of one RTT for the Quick-Start handshake. Furthermore, the used Quick-Start implementation rounds the requested data rate down to the closest allowed request rate. Because of both effects, it generally makes sense to request for a slightly smaller target server response time than actually desired. Then, Quick-Start also performs well.

The result for a real Web page is presented in Figure 7.5. For this experiment, the "OpenStreetMap" Web page has been mirrored on a local server. This Web page has been selected as an example for a page with a significant number of larger objects, such as the images of the map tiles. Furthermore, the interactive usage of such maps requires that the Website is almost as fast as a standalone application if it shall be able to replace an offline navigation solution. The server is accessed over a path with $r = 1\,\text{Gbit/s}$ and $\tau = 200\,\text{ms}$, i. e., there is plenty of bandwidth to download the page, which sums up to a total amount of data of about 1.5 MB. The Web page loading time T_{page} is determined by the "Firebug" extension of the "Firefox" browser and averaged over seven repeated experiments with a disabled browser cache.

In this setup, the page loading time with a default TCP stack is about 9.6 s. One reason for this rather large value is the design of the "OpenStreetMap" Web page: It must first download several large *JavaScript* files before the actual download of the map data can start. In such a situation fast startup schemes make sense. If the Web server uses either Jump-Start or Initial-Start, the loading time is reduced by over one second, independent of the signaled target time. For a human user, this reduction results in a perceivably faster rendering of the Web page.

A further reduction is possible with Quick-Start or Mega-Start: If the target response time is set to a value smaller than 300 ms, the *page loading time is reduced by over two seconds*. In the given scenario, the optimum value both for Quick-Start and Mega-Start is $T_{\text{target}} \approx 250\,\text{ms}$. As the given path has a large capacity, both Quick-Start and Mega-Start can then use a rather large sending rate and thus minimize transfer delays. In contrast, if the response time target is of the order of one second, the determined initial rate is rather small, and there is no benefit compared to TCP's standard Slow-Start. If the target value T_{target} is very close to the RTT of 200 ms, the page loading time is also slightly worse than the optimum. According to Equation (7.1), the used data rates are then very high, and the activation of the fast startup becomes sensitive to jitter in the RTT estimation. As already explained, the Web server in this experiment only activates the fast startup if the target response time is smaller than the RTT, i. e., a small increase of the estimated RTT can prevent the activation.

In summary, this case study shows that fast startup schemes perceivably improve the page loading time of a Website like "OpenStreetMap". The loading time can be reduced by up to several seconds. The advantage of schemes with explicit activation is that the download speed can be influenced by the user, which can selectively decide on the urgency. In contrast, the implicit schemes result in an undifferentiated speedup. It is an open issue whether an additional interface would be acceptable to users, i. e., if users would be interested in having an impact on the aggressiveness of the congestion control and the speed of the Web, e. g., by a simple button in the Web browser as shown in Figure 7.3. It would require large-scale field test with representative groups of users to finally decide whether a signaling solution indeed makes sense, or whether TCP's existing model of not offering any control continues to be the preferred solution.

7.2 Speedup of 3D visualization applications

7.2.1 Network challenges of interactive 3D applications

More and more applications interactively display *three-dimensional* (3D) content and provide rich visualizations and graphical renderings of real or virtual worlds and foreshadow a new class of networked, multi-purpose, interactive 3D applications. Many ongoing research and development activities are driven by the vision that it will be possible to create a new "World Wide Space" [125] or "3D Web" [128] that offers ubiquitous access to detailed representations of the real or virtual world and that combine realistic 3D city models with other data sources, location-based services, and potentially even live content. These applications are enabled by several technical developments: Hardware-accelerated 3D processing is very common even on mobile devices, and more and more 3D content is available, both from commercial providers and from communities, and there is also an increasing number of solutions to solve the interoperability between heterogeneous data sources. A detailed discussion of these technological enablers and the different possible architectures of interactive 3D applications is outside the focus of this thesis. These aspects are surveyed in reference [207].

Many emerging *online* 3D applications are client-server applications with *network-demanding communication characteristics*. The bandwidth requirements of interactive 3D applications can be very high, since voluminous content must be transported over the Internet in a timely fashion. These communication patterns are distinct from other application classes such as classic Web or linear multimedia streaming applications. The encoding of complex 3D structures can sum up to large amounts of data, in particular if it is encoded in XML and if the model also includes high-quality textures and aerial images. Examples presented in [207] show that even a simple 3D city scenario results in a data volume of the order of one Megabyte. The size of the images of building textures can easily exceed $10MB$. This communication volume can of course be reduced by techniques introduced in Section 3.2, such as *caching* and *loss-less compression*. Furthermore, 3D visualization applications also have a multitude of possibilities to adapt their communication behavior by controlling the *level of detail* of the visualized content as well as other visualization parameters [26]. But the description of a complex 3D scenery is still orders of magnitude larger than a typical Website. 3D models are thus one example for "mid-sized" data transfers which can be significantly speeded up by fast startup congestion control. Also, assuring the *responsiveness* of the application is a critical usability requirement, since visualization lags can be immediately noticed by users.

Unlike other linear multimedia streaming traffic, the data delivery of most 3D model parts has to be reliable [26]. This is why most online 3D visualization applications either use HTTP or proprietary application protocols on top of TCP, even if various UDP-based application protocols have also been developed (see [207]). One reason is that TCP is not optimized for both reliable *and* timely transport of data. In this context, the delays caused by the Slow-Start have been identified as a problem, e. g., in reference [166]. Furthermore, applications require knowledge about the network characteristics in order to adapt and optimize their 3D visualization, in particular an estimate how long the transport of a certain amount of data will probably last [26].

In summary, browsing interactively through complex, high-quality 3D scenarios retrieved from the Internet results in large and bursty data downloads triggered by user actions. Because of the combination of this workload characteristics and the requirement of responsiveness, online 3D visualization applications are an obvious use case for fast startup mechanisms.

7.2.2 Architecture of an interactive 3D visualization application

Within the Nexus project at the University of Stuttgart [125], interactive visualization applications for 3D city models are developed. One of these prototypical applications has been tested in combination with the Linux network stack implementations of fast startup congestion control schemes. The selected application has the advantage that it is Linux-based and that it offers full access to the source code of both client and server, which facilitates instrumentation and experiments in controlled environments. The client application can visualize 3D city models similar to other commercial tools, such as "Google Earth". In order to retrieve the model *scenegraph*, the client periodically synchronizes its context with the server as illustrated in Figure 7.6. A

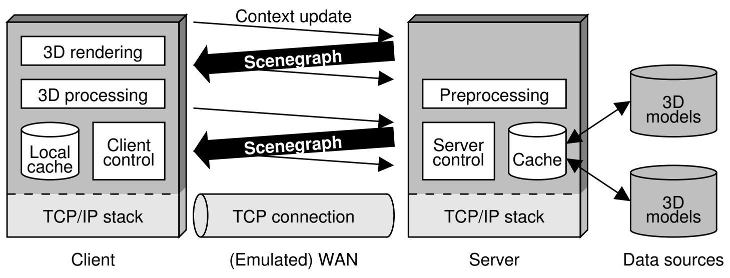


Figure 7.6: Client-server architecture of an interactive 3D visualization application

scenegraph is a graph structure that represents a 3D environment. If the user changes the visualization perspective, the missing 3D model parts are retrieved from the server. In the prototype implementation, the data transport is realized over a single persistent TCP connection using a binary encoding format. The architecture is open to other data sources, as the server can query different external databases and retrieve various 3D models that are available in XML formats.

In the following experiments, the client and the server are interconnected again by an 1 Gbit/s Ethernet segment. This represents a capacity over-provisioning scenario, since the applications cannot process the content at this speed. The Linux network emulation "NetEm" enforces a minimum RTT of 200 ms in order to realize a scenario in which the servers providing the 3D models are distributed around the world and accessed by mobile devices, as envisioned by the Nexus project.

Client and server applications have been instrumented in order to measure the response times, being defined as the time between the sending of a context update of the client and the complete reception of the corresponding new scenegraphs as illustrated in Figure 7.6. Furthermore, the communication is analyzed by capturing "tcpdump" traces. In order to reproduce the results, the measurement uses a recorded sequence of user interactions that browse through a couple of different locations where 3D city models are available. At the beginning of each measurement, the client cache is empty, i. e., the complete 3D scenery must be retrieved from the server. This represents a typical scenario in which a user explores a new part of the 3D world.

7.2.3 Exemplary measurement results

Figure 7.7 shows one example of a resulting trace. The upper part of the diagram depicts the observed data rate between server and client as a function of the time in the recorded sequence of user interactions. The lower part reports the response times measured by the client application. Each measurement point refers to the delay between the complete download of a new 3D scenegraph object and the client message that triggered that transfer. Obviously, as long as the 3D scenery does not change, no data transfers occur. Due to the size of the displayed 3D models, the total amount of data exchanged during one measurement run is large. For instance, after about 150 s, a city model of Frankfurt is loaded and displayed, which includes many buildings. All objects of this model part sum up to more than 80 MB.

In Figure 7.7 one can observe the typical TCP behavior, i. e., the sender starts to send with a small data rate, which then ramps up exponentially. As shown in the lower part, it can last up to 20 s until all parts of a new 3D model are retrieved, even though the link has a vast amount of free capacity. This proves that the CUBIC congestion control is apparently sub-optimal for this communication pattern.

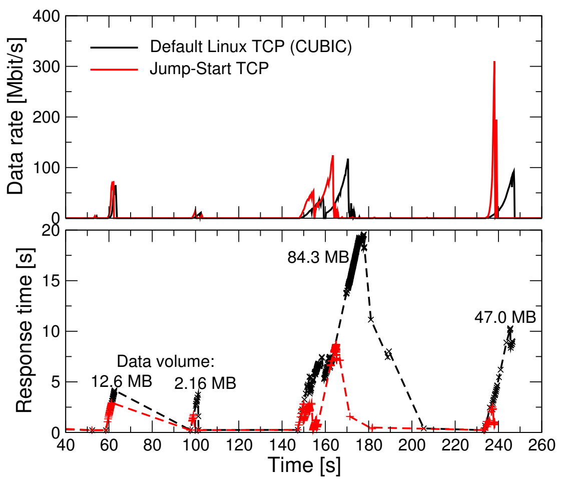


Figure 7.7: Traces from a 3D visualization tool with Slow-Start vs. Jump-Start

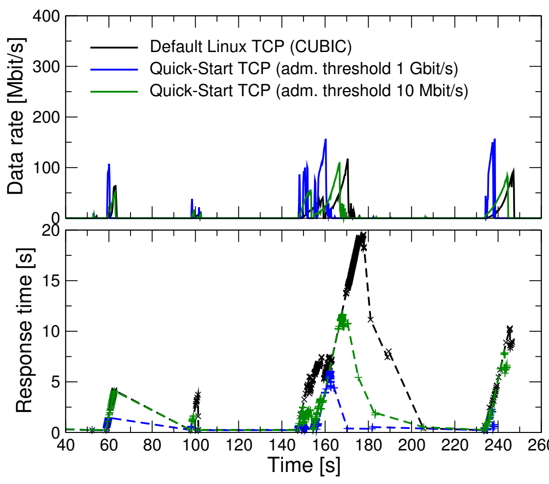


Figure 7.8: Traces from a 3D visualization tool with Slow-Start vs. Quick-Start

The performance of Jump-Start is also included in Figure 7.7. In the diagram, the peaks of the fast startups are clearly visible. In this experiment, the usage of Jump-Start significantly reduces the response times whenever data is transferred. In the best case, the model loading delay is reduced by over 10 s, which is a very significant reduction.

The same experiment has also been performed with the Quick-Start TCP extension. Figure 7.8 shows results for two approval control configurations: In the first case it is assumed that the total link capacity is available to Quick-Start requests ($c = 1$ Gbit/s, $\theta = 1$), while in the second case the threshold of the target algorithm is only $\theta = 0.01$, i. e., only 10 Mbit/s are available. In both setups, the requested initial sending rate is 82 Mbit/s, which is a reasonable value to transfer even large models within few seconds.

If the admission threshold is high, all Quick-Start requests throughout the experiment are approved, i. e., the initial sending rate is always of the order of 100 Mbit/s. Figure 7.8 shows that in this case the content download times are reduced to few seconds only and even smaller than the Jump-Start results. If the admission threshold is only 10 Mbit/s, the Quick-Start requests are reduced in the network stack. The corresponding results in Figure 7.8 still show a performance improvement. This reveals that it is sufficient to grant only a certain share of the link capacity to Quick-Start requests. But, of course, the maximum achievable speedup is then smaller.

Similar experiments have also been realized with other high-speed congestion control algorithms and resulted in similar graphs. In experiments with smaller RTTs such as 50 ms, the potential response time reduction by a fast startup scheme is smaller and hardly exceeds one second. This result reconfirms that fast startups are mainly beneficial for transport over long-distance paths. One could argue that large RTTs could be avoided by hosting the content inside a CDN, as explained in Section 3.2.2.3. However, as already mentioned, it is very unclear whether CDNs will indeed be able to deliver complex and potentially dynamic 3D models, in particular once they get synchronized with the real world.

A shortcoming of the presented experiment is that the prototypical application cannot use multiple parallel TCP connections. Nevertheless, other online 3D visualization applications have similar workload characteristics and would thus benefit from fast startups, too.

8 Conclusion

8.1 Summary

The Internet can never be fast enough. The TCP congestion control has been a key to the Internet's operational success in the last decades. Congestion control efficiently realizes best effort transport and continues to be the predominantly used resource management principle in packet networks, despite a vast amount of research on network QoS mechanisms. All widely used TCP implementations use the Slow-Start algorithms, which is a fundamental principle of the current Internet congestion control. However, the Slow-Start heuristic is time-consuming, delays the transport of larger amounts of data and is therefore not an ideal solution for broadband interactive applications. This naturally raises the question if and how faster startups could be realized, in particular over paths that traverse long-distance WANs or cellular access networks.

This question is one of the remaining fundamental challenges in the current Internet. As shown by the survey in this thesis, it is not new and has already been addressed by numerous studies in the past decades. It is well-known that the flow startup is a complex problem, and if there was a simple and robust alternative to the Slow-Start heuristic, it would probably be already in use today. Yet, research work in the last years has opened up new perspectives: On the one hand, new high-speed congestion control algorithms are more aggressive than TCP Reno. The philosophy behind their design is that congestion must not be minimized at all cost, given that today's TCP stacks very efficiently recover from packet loss. Following this trend, one could also replace the Slow-Start by a more aggressive startup. One the other hand, it is believed that in the future long-term evolution of the Internet additional signaling mechanisms along a path could be possible, in particular if no per-flow state is required in the network components. This vision has recently driven the development of new network-supported congestion control protocols that also ramp up data rates faster than the currently used algorithms.

This thesis follows both trends and comprehensively evaluates different *fast startup congestion control mechanisms* that can almost immediately utilize links even in high-speed networks. It considers both end-to-end schemes as well as protocols that use signaling along the path. Since it is impossible to test the complete design space of flow startup schemes, the main focus are novel, promising schemes that have been proposed in the last years, in particular the Jump-Start proposal and the Quick-Start protocol. Quick-Start is an experimental TCP extension that allows hosts to cooperate with the routers along a path in order to determine a large initial sending rate. Also, a new combination of both principles (Mega-Start) is proposed, and the trivial mechanism of just increasing the initial window is considered as well (Initial-Start). All these TCP enhancements are implemented in the network stack of the Linux operating system in order to enable realistic experiments. Their performance is studied by analytical models, by simulations, by measurements in local testbeds, as well as by some tests over a real network path. The study is complemented by a comparison to two other well-known protocols, XCP and

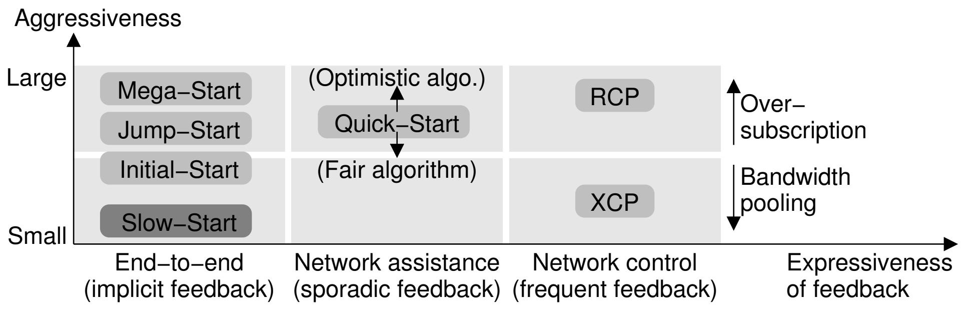


Figure 8.1: Classification of the congestion control mechanisms considered in this work

RCP, which have been developed as part of the ongoing clean slate research activities towards a Future Internet.

From a global point of view, these different congestion control schemes differ in two very fundamental aspects: First, the network feedback inherently has a larger *expressiveness* if signaling is used. The advantage of a large expressiveness is that more information is made available, but it also comes along with the risks: In an open network like the Internet, any information must be secured, and the resource management must be robust even if information is inaccurate. This thesis proves that network-controlled schemes such as XCP or RCP are very sensitive to inaccuracy of the information about the link capacity, whereas a network-assisted scheme such as Quick-Start is much more robust, since it only uses feedback with a coarse granularity and does not replace the end-to-end congestion control. Second, the schemes have different levels of *aggressiveness*. There is an inherent trade-off between the maximum possible speedup during a flow startup and the risk of congestion, independent of how the algorithms are designed in detail. This work shows that this trade-off even exists if a signaling protocol provides information about the path characteristics. Any resource management must either use an *oversubscription* or a *bandwidth pooling* strategy; in the former case, there is a larger risk of causing congestion, in particular if many new flows arrive in parallel, whereas in the latter case it is non-trivial to deal with traffic patterns that consist of many short-lived flows. In Figure 8.1, the schemes investigated in this work are classified along these two dimensions.

All fast startup schemes reduce the delays for larger transfers compared to Slow-Start, if they are properly used. Still, this work identifies also significant differences. The findings concerning the different schemes are summarized below. These advantages and drawbacks are also listed in Table 8.1.

The *Jump-Start* mechanism has a quite reasonable performance in many scenarios, if it is modified as proposed in this work. It achieves a reasonable speedup of mid-sized transfers at the cost of a moderately larger packet loss probability. While there are naturally cases in which the fast startup is too aggressive, in many investigated scenarios the increase of packet losses is of the order of 1 % only, which is hardly critical for a modern TCP implementation, and also does not necessarily result in significant performance degradation of other, standard-compliant TCP connections sharing the same bottleneck. Jump-Start uses rate pacing during the first RTT.

A simpler, though slower alternative is just to increase the initial Congestion Window (*Initial-Start*). This work experiments with an initial window of 10 MSS, and such a moderate increase does not seem to be overly harmful. A significantly larger increase of the initial window is not an option, as it would then for sure exceed typical buffer sizes in network components. But if TCP's initial window was increased by few segments only, more advanced fast startup schemes

Table 8.1: Summary of the properties of fast startup schemes

Scheme	**Advantages**	**Drawbacks**
End-to-end		
Jump-Start	- Speedup of larger transfers - Less aggressive over long paths	- Increased risk of packet losses - Behavior depends on application write patterns and socket buffer sizes
Initial-Start	- Very simple implementation - Reasonable speedup	- Increased risk of packet losses - Bursty traffic due to lack of rate pacing
Mega-Start	- Improved convergence to equal sharing of bandwidth - Sending rate adjustable by application	- Potentially very aggressive - Should not be used by all applications
Network-supported		
Quick-Start	- Low risk of causing congestion - Simple coexistence with end-to-end congestion control - Sending rate adjustable by application	- Deployment challenges of IP option processing - Intelligent activation heuristics required
XCP	- Operation without packet drops - Convergence to equal bandwidth sharing	- Vulnerable to capacity estimation errors - Slower than TCP Reno for short transfers - Isolation of TCP Reno traffic needed - Many open issues, e. g., concerning security
RCP	- Operation without packet drops - Transfer time close to processor sharing - Very fast convergence to equal bandwidth sharing	- Vulnerable to capacity estimation errors - Large buffers in network components required - Isolation of TCP Reno traffic needed - Incomplete protocol specification

can hardly achieve further benefits except in extreme situations. Both the Jump-Start proposal as well as an increased initial window benefit from the typical Internet traffic characteristics: Since many data transfers are rather small, faster startups only affect a smaller number of flows. This principle could also be paraphrased by the design philosophy of *making the rare case fast*. Both schemes also have in common that they could be a drop-in replacement of the currently used Slow-Start algorithms and could thus be used by all TCP data transfers without explicit support by the application.

As an alternative to a new Slow-Start algorithm, this thesis also proposes a solution that enables the fast startup only in selected cases, i. e., upon explicit activation by the application by a new interface (*Mega-Start*). Under the assumption that the application or another entity can provide a reasonable value for an initial rate that is roughly of the order of the available bandwidth on the path, the Mega-Start scheme can work rather well. It also has the interesting property that it allows a differentiation between different application types. Furthermore, applications then have some control over the expected data transfer times, which can improve the application-level QoS. However, the experiments in this work also show that such a potentially very aggressive startup could cause harm if it was used by all applications all the time. Its usefulness crucially depends on the ability of application developers or users, which must decide whether to use it, or whether the standard Slow-Start is just sufficient.

The fundamental challenge for all flow startup approaches is the lack of information about the characteristics of an unknown path. This uncertainty could be reduced by new signaling protocols. A major contribution of this thesis is the investigation of the *Quick-Start* protocol under realistic constraints. Given that the Quick-Start protocol only activates a fast startup if the routers along the path observe a sufficient available capacity, the risk of congestion is much smaller compared to end-to-end fast startup solutions. This works proves that Quick-Start is a lightweight network-assisted mechanism that can be implemented both in endsystems and routers with limited overhead, even though it suffers from the poor incremental deployment properties of all network-supported mechanisms. The practical experiments with Quick-Start in this thesis confirm the findings of previous simulation studies.

Furthermore, new algorithms are designed for the Quick-Start protocol. This thesis proposes two new approval control strategies in order to mitigate the inefficient allocation of bandwidth of the existing approval control algorithms: In case that the approval control of a router should be conservative and use the design philosophy of bandwidth pooling, the proposed *fair algorithm* ensures that the available bandwidth is more fairly assigned to different requests, without much additional algorithmic complexity. This thesis also shows that an alternative router strategy could be oversubscription, which makes the whole Quick-Start mechanism much simpler (*optimistic algorithm*) without significantly increasing the congestion risk unless flash crowd effects occur. In any case, the Quick-Start mechanism requires intelligence in the endsystems. They must decide when to issue a request, decide on a reasonable rate, and avoid querying for bandwidth that they will not use. If Quick-Start is just activated naively for all data transfers, the mechanism is hardly of any benefit, or it may even worsen the performance compared to the usage of Slow-Start. There are other non-trivial problems, such as the handling of the Slow-Start Threshold. A promising solution is to provide an interface by which applications can selectively activate the Quick-Start mechanism. The experiments in this work with real applications show that such an activation is indeed possible in practice.

Finally, this work compares Quick-Start to two other well-known network-controlled congestion control frameworks that have been proposed for a Future Internet: *XCP* and *RCP*. Compared to these disruptive, clean-slate approaches, Quick-Start is found to be a simple, evolutionary solution, since it is more robust and does not cause interoperability problems with flows using an existing end-to-end congestion control. In the currently proposed form, XCP and RCP cannot be used for Internet congestion control. Despite numerous work on XCP, the algorithms are not suitable for typical Internet workloads. The performance results of the more recent RCP are more promising. In fact, Quick-Start with an approval control using the optimistic algorithm has some similarities with RCP. Yet, many open issues of RCP remain unsolved. This thesis shows that, unlike Quick-Start, XCP and RCP crucially depend on precise information about link capacities, which can hardly be obtained in current network architectures. According to these results, if a network-supported fast startup congestion control were to be deployed in the Future Internet or, alternatively, in controlled intranets, Quick-Start would be the only candidate out of the currently known techniques.

A further contribution of this work is to show-case the applicability and benefits of fast startup congestion control with real applications. Fast startup congestion control is only useful for selected classes of applications, which are classified as *broadband interactive applications* in this work. Their key characteristic is that they are delay-sensitive and that they frequently exchange mid-sized or larger amounts of data. There are various different examples for such use cases. This thesis substantiate that such characteristics are common in applications that deal with two-dimensional or three-dimensional representations of the real world. Emerging online 3D visualization applications exhibit communication patterns that are distinct from other application classes such as classic Web or video streaming applications, and are thus a typical representative of broadband interactive applications. The presented case studies illustrate that such network-challenging applications could significantly benefit from fast startup congestion control. Depending on the scenario, speedups of up to several seconds can be achieved. Additionally, the case studies show that fast startup congestion control could be the missing piece in an overall network architecture in which applications could specify transport delay constraints, e. g., in order to meet certain response time targets. Such a mechanism could become important as more and more Internet applications replace locally installed software and must thus interact with users with very small delays.

In summary, this thesis shows that fast startup congestion control would be a *promising mechanism in the future evolution of the Internet* and would help to overcome one of the few remaining performance limitations. A general observation in the experiments is that both end-to-end and network-supported fast startup schemes trade off the speedup and the risk of congestion, even though network support can limit this risk. But if one is willing to accept a moderately increased packet loss probability, one can question whether the complexity of network support is indeed worth the effort. In average, a sophisticated end-to-end fast startup like Jump-Start can achieve quite similar performance results. The results of this work can thus be summarized by three key conclusions:

1. End-to-end fast startup schemes are less harmful than often assumed, if they are carefully designed and selectively used.
2. Network-support can overcome inherent limitations of any end-to-end congestion control, but it has problems of its own, and requires intelligent usage strategies even if the deployment issues were solved.
3. The key challenge is the cross-layer interaction with applications, which could be addressed by additional interfaces to the network stack.

8.2 Outlook

This thesis is closely related to the fundamental question of how the Internet congestion control should evolve in future. This question is addressed by many ongoing discussions and research activities, but it is far from being solved. Many of the open research issues are surveyed in Section 2.4.3, Section 4.2.5, and Section 4.7.1.2. Concerning fast startup congestion control, there are several challenges that require further research, as they are not completely solved by this thesis (see also [172]):

- A fundamental question is whether a slightly increased packet loss probability is acceptable in the Internet if this improves the overall application performance. A recent proposal of a major Internet application provider [49] also argues in favor of more aggressive TCP algorithms and faster startups, but the question remains controversial.
- There is a lack of theoretical models for understanding and evaluating flow startup mechanisms, in particular concerning their impact on congestion risk, stability, and fairness. This thesis studies the resulting trade-offs in various scenarios, but in fact an evaluation in very large network topologies would be required, too.
- This thesis argues in favor of differentiated starting schemes, i. e., certain classes of applications can use higher initial sending rates. Such a differentiation inherently raises fairness issues. Currently, there is no established theoretical methodology to quantify the

fairness of flow startup schemes. Another question is how incentives could be created so that aggressive startup schemes are only used when they are indeed useful. Such an incentive solution could be designed based on the work of Briscoe [32, 35].

- The challenge for any congestion control scheme is how to deal with highly dynamic changes of the path characteristics. Further work is required to understand the trade-offs of end-to-end fast startup mechanism vs. network support in such an environment, in which the feedback obtained from the network may be very inaccurate. It is unclear whether the signaling is then indeed worth the effort.
- One could design congestion control signaling protocols other than the ones investigated in this thesis. Yet, there is still no common understanding about the syntax and semantics of such a protocol, which would be a candidate for replacing TCP in a "Post-TCP era" [198]. Any fundamentally new Internet transport mechanism will likely come along with a new flow startup scheme.

The studies in this thesis show that the startup of a data transfer is not a sole congestion control problem, but instead heavily depends on the interaction with applications. These application issues can only partly be addressed in this thesis. Applications may have to be redesigned in order to get the best benefit of fast startup congestion control, and new application-network interfaces might be needed. This thesis proposes and showcases such interfaces, but it cannot study all implications. Topics of particular interest are *adaptive applications* that can change their communication behavior depending on the network characteristics (cf. [208]). If an application has an own control logic, this could facilitate the configuration of a fast startup scheme according to the application's demand. But then actually a further interface would make sense, which exposes information about the network characteristics, such as the available bandwidth or an estimate how long the transport of a certain amount of data will probably last. These interfaces could also be enhanced to provide feedback from fast startup congestion control mechanisms. For instance, the approved Quick-Start rate could be reported to applications so that they can adapt accordingly. However, the interaction of fast startup congestion control and adaptive applications is not further investigated in this work, because adaptation decisions can also be realized without information from the network stack. A further possibility would be cross-layer adaptation interfaces where the network stack can trigger adaptations of context-aware applications [60]. The handling of such triggers inside applications is still a research topic.

There are also certain specific design choices of fast startup congestion control schemes that could be studied in future work:

- There are several different possibilities how a fast startup scheme could react to lost or marked packets. It can use TCP's standard recovery algorithm (used, e. g., in Initial-Start), modify the CWND by a new algorithm (e. g., Jump-Start, Mega-Start), or revert to the previous state (e. g., Quick-Start). The response after packet loss is actually orthogonal to the initial ramp up of the CWND, i. e., there could be other combinations or algorithms as well. These algorithmic details are left for further study.
- The experimental results do not allow to finally conclude whether a fast startup indeed needs *rate pacing* if the burst size is of the order of ten packets only. Not using rate pacing obviously increases the burstiness of the traffic. The impact of bursts significantly depends on the characteristics of the buffers in network components, which vary extremely. In general, it is impossible to model the buffer characteristics in the Internet, so that this question can only be addressed by experiments with real equipment.

A remaining open issue of this thesis are experiments in large-scale networks. The overall implications of using fast startup congestion control on Internet scale can only be understood by actual deployments. The author has started to perform experiments in an experimental network testbed, but this infrastructure was then not available any more. Other large-scale experimental platforms are still under construction. Currently, it is not possible to study network-supported congestion control schemes in the existing experimental platforms, as hardware implementations in the network components are required. End-to-end congestion control schemes could be tested, and this would be the most obvious next step of this work. The availability of implementations in the Linux network stack facilitates such experiments. Experimental studies with fast startup congestion control schemes are also planned in the workplan of upcoming experimental Future Internet platforms [230]. Such experimental work is required since simulation studies are not sufficient to evaluate protocol mechanisms at Internet scale.

A Appendix: Mathematical background

A.1 Distribution functions

Performance evaluation uses statistical methods. In this work, several distribution functions are used. The distributions can be described by their probability density function $f(z)$ or by their distribution function

$$F(z) = P(Z \leq z) = \int_0^z f(t)\,\mathrm{d}t. \tag{A.1}$$

Well-known distributions are the *exponential* or *negative-exponential distribution*, the *pareto distribution*, and the *log-normal distribution*.

A.1.1 Exponential distribution

The exponential distribution is characterized by

$$F_{\text{negexp}}(z) = \begin{cases} 1 - \exp\left(-\lambda_{\text{negexp}} \cdot z\right) & z \geq 0 \\ 0 & z < 0 \end{cases}. \tag{A.2}$$

The mean value of the exponential distribution is $m_{\text{negexp}} = 1/\lambda_{\text{negexp}}$.

A.1.2 Pareto distribution and variants

The pareto distribution function is defined as

$$F_{\text{pareto}}(z) = \begin{cases} 1 - \left(\frac{z}{k_{\text{pareto}}}\right)^{-\alpha_{\text{pareto}}} & z \geq k_{\text{pareto}} \\ 0 & z < k_{\text{pareto}} \end{cases} \tag{A.3}$$

with shape parameter α_{pareto} and location parameter k_{pareto}. The mean is $m_{\text{pareto}} = \frac{\alpha_{\text{pareto}} \cdot k_{\text{pareto}}}{\alpha_{\text{pareto}} - 1}$ for $\alpha_{\text{pareto}} > 1$. The mean value is infinite for $\alpha_{\text{pareto}} \leq 1$, and the distribution has an infinite variance for $\alpha_{\text{pareto}} \leq 2$. Because of the these properties, the *truncated pareto distribution* is often used. Due to the truncation at $z = K_{\text{pareto}}$, the distribution function is $F_{\text{tpareto}}(z) = F_{\text{pareto}}(z)/(1 - (\frac{K_{\text{pareto}}}{k_{\text{pareto}}})^{-\alpha_{\text{pareto}}})$ for $k_{\text{pareto}} \leq z \leq K_{\text{pareto}}$. The mean value of the truncated pareto distribution is $m_{\text{tpareto}} = \frac{\alpha_{\text{pareto}} \cdot k_{\text{pareto}}}{\alpha_{\text{pareto}} - 1}(1 - (\frac{K_{\text{pareto}}}{k_{\text{pareto}}})^{-\alpha_{\text{pareto}}+1})/(1 - (\frac{K_{\text{pareto}}}{k_{\text{pareto}}})^{-\alpha_{\text{pareto}}})$.

A.1.3 Log-normal distribution

The log-normal distribution has a probability density function of

$$f_{\mathrm{lognorm}}(z) = \begin{cases} \frac{1}{\sqrt{2\pi}\cdot\sigma_{\mathrm{lognorm}}\cdot z} \exp\left(-\frac{\left(\ln z - \mu_{\mathrm{lognorm}}\right)^2}{2\,\sigma^2_{\mathrm{lognorm}}}\right) & z > 0 \\ 0 & z \leq 0 \end{cases}. \tag{A.4}$$

The mean value is given by $m_{\mathrm{lognorm}} = \exp\left(\mu_{\mathrm{lognorm}} + \sigma^2_{\mathrm{lognorm}}/2\right)$.

A.2 Analytical lower bound for TCP resequencing delays

A.2.1 Scope of the model

An analytical model for the TCP resequencing delays has been published in [197, 116]. This section briefly summarized the findings. It only considers the case of a single TCP connection and does not investigate the performance of well-known techniques to mitigate the impact of Head-of-Line blocking, which include SCTP's multi-streaming or the unordered mode [116], or, alternatively, multiple parallel TCP connections between the same two endsystems [197].

A.2.2 Model details

According to [197, 116], the average resequencing delay caused by HOL can be explicitly calculated if the minimum RTT τ and the packet loss probability p is known, and if the impact of the congestion control can be neglected. This is a reasonable assumption as long as p is small. TCP can recover from packet loss either by the *fast retransmit* mechanism or by the *retransmission timeout*. In the former case, the sender can detect the packet loss after the arrival of three duplicate ACKs, i. e., after a delay of $\tau + 3\,\delta$. For simplicity, in this expression it is supposed that signaling messages are sent with constant IAT δ. The error detection time by the Retransmission Timeout, which is restarted whenever a new ACK arrives, is $T_{\mathrm{o}} + \max(\tau - \delta, 0)$. Considering both mechanisms, the error detection time is

$$T_{\mathrm{det}} = \min\left(\tau + 3\,\delta, T_{\mathrm{o}} + \max(\tau - \delta, 0)\right). \tag{A.5}$$

This expression is an approximation only since more than one packet, the retransmission, or acknowledgments may get lost. Lost ACKs could be handled by a simple extension [116]. In Linux stacks the RTO duration T_{o} is typically close to its minimum value $200\,\mathrm{ms} + \tau$ (see [189]).

The impact of resequencing delays on the response time can than be calculated as follows: TCP segments have to wait in the receiver's resequencing queue until the retransmission arrives. The waiting times ω_i depend on the time $T_{\mathrm{det}} = \omega_0$ to detect the packet loss. The number of segments that have to be queued until the retransmission arrives is $N_{\mathrm{queued}} = \lfloor T_{\mathrm{det}}/\delta \rfloor$. The resequencing delay of the i-th segment after the lost one is $\omega_i = T_{\mathrm{det}} - i \cdot \delta$. The mean waiting time is the sum of all ω_i divided by the mean number of segments between two losses, which is $1/p$:

$$T_{\mathrm{hol}} = p \sum_{i=0}^{N_{\mathrm{queued}}} \omega_i = p \left((N_{\mathrm{queued}} + 1) \cdot T_{\mathrm{det}} - \left(N_{\mathrm{queued}}\,(N_{\mathrm{queued}} + 1)\right) \frac{\delta}{2} \right). \tag{A.6}$$

Since HOL may occur in both directions and since the additional processing time in the endsystems ε has to be considered, the mean response time follows as

$$T_{\mathrm{resp}} = \tau + 2\,T_{\mathrm{hol}} + \varepsilon. \tag{A.7}$$

B Appendix: Documentation of parameters

This appendix documents the parameters of various experiments presented in this document.

B.1 Measurement setups

The measurement computers used in laboratory experiments have the following properties:

Component	Type	Comment
CPU	Intel Pentium 4 2.8 GHz	In Section 7 also Intel Core 2
Memory	2 – 4 GiB	
Network interface card	Intel Gigabit Ethernet	Connected to a PCI bus
Operating system	Ubuntu 7.04	In Section 7 also Ubuntu 8.04
Linux kernel version	2.6.24	See also Section 6.1.1.2

The computers used in the measurements over the experimental Internet path use a Fedora Linux operating system with Linux kernel 2.6.24 and have a different hardware.

B.2 General Linux kernel configuration

Both in measurements and simulations, the following Linux kernel system configuration parameters are used by default, unless mentioned otherwise:

Parameter	Default	Comment
`net/core/rmem_default`	8388608	System-wide default receive buffer size
`net/core/rmem_max`	16777216	System-wide maximum receive buffer size
`net/core/wmem_default`	8388608	System-wide default send buffer size
`net/core/wmem_max`	16777216	System-wide maximum send buffer size
`net/ipv4/tcp_rmem`	16384	Auto-tuning: Minimum receive buffer size
	8388608	Auto-tuning: Initial receive buffer size
	16777216	Auto-tuning: Maximum receive buffer size
`net/ipv4/tcp_wmem`	16384	Auto-tuning: Minimum send buffer size
	8388608	Auto-tuning: Initial send buffer size
	16777216	Auto-tuning: Maximum send buffer size
`net/ipv4/↩ tcp_no_metrics_save`	1	Connection statistics caching disabled

Parameter	**Default**	**Comment**
`net/ipv4/tcp_frto`	0	Known implementation error in kernel version 2.4.24 (cf. Section 6.1.1.2)

Furthermore, the socket option `TCP_NODELAY` is enabled in order to avoid any interaction with the Nagle algorithm.

B.3 Default configuration of the fast startup schemes

B.3.1 Jump-Start

Parameter `net/ipv4/...`	**Default**	**Comment [possible values]**
`tcp_js_enable`	1	1: enabled [0,1]
`tcp_js_max_size`	65535	Upper threshold in byte [>0]
`tcp_js_pacing_chunk`	3	Rate pacing chunk size in MSS [>0]
`tcp_js_ignore_rwnd`	10000000	Rate limit in byte/s; RWND is ignored in the first RTT for any positive value [>0]

B.3.2 Initial-Start

Parameter `net/ipv4/...`	**Default**	**Comment [possible values]**
`tcp_js_enable`	1	1: enabled [0,1]
`tcp_js_size`	10	Initial window in MSS [>0]
`tcp_js_ignore_rwnd`	1	Ignore RWND [0,1]

B.3.3 Mega-Start

Parameter `net/ipv4/...`	**Default**	**Comment [possible values]**
`tcp_js_enable`	1	1: enabling by sockets interface; 2: automatic activation by kernel [0,1,2]
`tcp_js_default_rate`	10000000	Default initial data rate when automatically activated [$\geq$0]
`tcp_js_pacing_chunk`	3	Rate pacing chunk size in MSS [>0]
`tcp_js_ignore_rwnd`	1	Ignore RWND in first RTT [0,1]
`tcp_js_synack_`↩ `workaround`	0	Sending of an additional <ACK> after the <SYN,ACK> (cf. Section 5.2.3) [0,1]

B.3.4 Quick-Start

Parameter `net/ipv4/...`	**Default**	**Comment [possible values]**
`ip_qs_enable`	1	$\leq$0: disabled; 1: enabled; 2: optimistic algorithms [0,1,2]
`ip_qs_link_capacity`	True value	Link capacity assumed by Quick-Start [$\geq$0]
`ip_qs_traffic_thresh`	100	Percentage that is used by approval control

Parameter `net/ipv4/...`	**Default**	**Comment [possible values]**
`ip_qs_traffic_interval`	0	0: automatic measurement of control interval; > 0: fixed configured value [$\geq$0]
`ip_qs_max_allowed_rate`	1310720000	≥ 0: Upper bound on granted rate in bit/s; -1: fairness enforcement [$\geq$-1]
`ip_qs_tracing`	0	Tracing of approval control state to the system log file [0,1]
`tcp_qs_enable`	1	0: disabled; 1: enabling by sockets interface; 2: automatic early activation; 5: automatic late activation [0,1,2,5]
`tcp_qs_idle_reactivate`	1	Retry a request on a path even if previous requests have been denied [0,1]
`tcp_qs_default_rate`	10000000	Default initial data rate when automatically activated [$\geq$0]
`tcp_qs_pacing_chunk`	3	Rate pacing chunk size in MSS [>0]
`tcp_qs_adapt_ssthresh`	0	0: QS-a; 2: QS-c; 3: QS-b [0,2,3]
`tcp_qs_ssthresh_factor`	1	Multiplication factor for SST in case of QS-b and QS-c [>0]
`tcp_qs_synack_`↩ `workaround`	0	Sending of an additional <ACK> after the <SYN,ACK> (cf. Section 5.2.3) [0,1]

Bibliography

[1] F. Abrantes and M. Ricardo. XCP for shared-access multi-rate media. *ACM SIGCOMM Computer Communication Review*, 36(3), pp. 27–38, 2006.

[2] B. O. M. Adjibadji. Untersuchung von Bandbreiteschätzverfahren hinsichtlich der Anwendung in Mobilfunksystemen. Student thesis (in German), Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2007.

[3] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *Proc. IEEE INFOCOM*, volume 3, pp. 1157–1165, 2000.

[4] V. Aggarwal, A. Feldmann, and C. Scheideler. Can ISPs and P2P users cooperate for improved performance? *ACM SIGCOMM Computer Communication Review*, 37(3), pp. 29–40, 2007.

[5] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proc. ACM SIGCOMM*, pp. 263–274, 1999.

[6] M. Allman. TCPx2: Don't fence me in. IETF Internet Draft, work in progress, May 2006.

[7] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated Quality-of-Service in Web hosting services. In *ACM SIGMETRICS Workshop on Internet Server Performance*, 1998.

[8] W. Almesberger. UML simulator. In *Proc. Ottawa Linux Symposium*, 2003.

[9] L. Andrew. Compound TCP in Linux. http://netlab.caltech.edu/lachlan/ctcp/.

[10] L. Andrew, C. Marcondes, S. Floyd, L. Dunn, R. Guillier, W. Gang, L. Eggert, S. Ha, and I. Rhee. Towards a common TCP evaluation suite. In *Proc. PFLDnet*, 2008.

[11] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *ACM SIGCOMM Computer Communication Review*, 34(4), pp. 281–292, 2004.

[12] J. Arkko, B. Briscoe, L. Eggert, A. Feldmann, and M. Handley. Dagstuhl perspectives workshop on end-to-end protocols for the future internet. *ACM SIGCOMM Computer Communication Review*, 39(2), pp. 42–47, 2009.

[13] M. Arns. A new aggregation technique for the analysis of extended open fork/join queueing networks by decomposition. In *Proc. MMB*, pp. 417–436, 2006.

[14] J. Aweya. On the design of IP routers part 1: Router architectures. *Journal of Systems Architecture*, 46(6), pp. 483–511, 2000.

[15] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. *ACM SIGMETRICS Performance Evaluation Review*, 26(1), pp. 151–160, 1998.

[16] P. Barford and M. Crovella. Critical path analysis of TCP transactions. *IEEE/ACM Transactions on Networking*, 9(3), pp. 238–248, 2001.

[17] L. A. Barroso, J. Dean, and U. Hölzl. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23, pp. 22–28, 2003.

[18] A. Bavier, L. Peterson, J. Brassil, R. McGeer, D. Reed, P. Sharma, P. Yalagandula, A. Henderson, L. Roberts, S. Schwab, R. Thomas, E. Wu, B. Mark, B. Zhao, and A. Joseph. Increasing TCP throughput with an enhanced internet control plane. In *Proc. IEEE MILCOM*, 2006.

[19] S. M. Bellovin, D. D. Clark, A. Perrig, and D. Song. A clean-slate design for the next-generation secure Internet. Report of a workshop of the National Science Foundation, 2004.

[20] S. Ben Fredj, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: A study of congestion at flow level. *ACM SIGCOMM Computer Communication Review*, 31(4), pp. 111–122, 2001.

[21] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5), pp. 64–71, 1999.

[22] E. Blanton and M. Allman. On the impact of bursting on TCP performance. In *Proc. Passive and Active Measurement Workshop (PAM)*, 2005.

[23] R. Bless and M. Doll. Integration of the FreeBSD TCP/IP-stack into the discrete event simulator OMNet++. In *Proc. Winter Simulation Conference*, 2004.

[24] S. Bodamer. *Verfahren zur relativen Dienstgütedifferenzierung in IP-Netzknoten*. PhD thesis (in German), University of Stuttgart, 2004.

[25] S. Bodamer, K. Dolzer, C. Gauger, M. Barisch, and M. Necker. *IKR Simulation Library 2.6 User Guide*. IKR, University of Stuttgart, 2007.

[26] I. M. Boier-Martin. Adaptive graphics. *IEEE Computer Graphics and Applications*, 23(1), pp. 6–10, 2003.

[27] O. Boxma, G. Koole, and Z. Liu. Queueing-theoretic solution methods for models of parallel and distributed systems. In O. Boxma and G. Koole, editors, *Performance Evaluation of Parallel and Distributed Systems - Solution Methods*. CWI (Tract 105 & 106), Amsterdam, 1994.

[28] L. S. Brakmo and L. L. Peterson. TCP Vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communication*, 13(8), pp. 1465–1480, 1995.

[29] L. Breslau, S. Jamin, and S. Shenker. Comments on the performance of measurement-based admission control algorithms. In *Proc. IEEE INFOCOM*, pp. 1233–1242, 2000.

[30] L. Breslau, E. W. Knightly, S. Shenker, I. Stoica, and H. Zhang. Endpoint admission control: Architectural issues and performance. *ACM SIGCOMM Computer Communication Review*, 30(4), pp. 57–69, 2000.

[31] P. G. Bridges, G. T. Wong, M. Hiltunen, R. D. Schlichting, and M. J. Barrick. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking*, 15(6), pp. 1254–1265, 2007.

[32] B. Briscoe, A. Jacquet, C. Di Cairano-Gilfedder, A. Carla, A. Soppera, and M. Koyabe. Policing congestion response in an internetwork using re-feedback. *ACM SIGCOMM Computer Communication Review*, 35(4), pp. 277–288, 2005.

[33] B. Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review*, 37(2), pp. 65–74, 2007.

[34] B. Briscoe, A. Jacquet, T. Moncaster, and A. Smith. Re-ECN: Adding accountability for causing congestion to TCP/IP. IETF Internet Draft, work in progress, July 2008.

[35] B. Briscoe. A fairer, faster Internet protocol. *IEEE Spectrum*, Dec. 2008, pp. 38–43, 2008.

[36] S. Bryant, B. Davie, L. Martini, and E. Rosen. Pseudowire congestion control framework. IETF Internet Draft, work in progress, June 2009.

[37] L.-O. Burchard. Analysis of data structures for admission control of advance reservation requests. *IEEE Transactions on Knowledge and Data Engineering*, 17(3), pp. 413–424, 2005.

[38] L. Burgstahler and M. Neubauer. New modifications of the exponential moving average for bandwidth estimation. In *15th ITC Specialist Seminar*, 2002.

[39] C. Caini and R. Firrincieli. TCP Hybla: a TCP enhancement for heterogeneous networks. *International Journal of Satellite Communications and Networking*, 22, pp. 547–566, 2004.

[40] G. Camarillo and M.-A. García-Martín. *The 3G IP Multimedia Subsystem (IMS): Merging the Internet and the Cellular Worlds*. Wiley, 2nd edition, 2005.

[41] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys*, 34(2), pp. 263–311, 2002.

[42] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proc. IEEE INFOCOM*, volume 3, pp. 1742–1751, 2000.

[43] V. G. Cerf and R. E. Kahn. A protocol for packet network interconnection. *IEEE Transactions on Communications*, 22(5), pp. 637–648, 1974.

[44] S. Chang and A. Vetro. Video adaptation: Concepts, technologies, and open issues. *Proceedings of IEEE*, 93(1), pp. 148–158, 2005.

[45] T.-Y. Chang, Z. Zhuang, A. Velayutham, and R. Sivakumar. Webaccel: Accelerating Web access for low-bandwidth hosts. *Computer Networks*, 52(11), pp. 2129–2147, 2008.

[46] J. Charzinski. Measured HTTP performance and fun factors. In *Proc. 17th International Teletraffic Congress (ITC)*, pp. 1063–1074, 2001.

[47] Z. Chen, T. Bu, M. Ammar, and D. Towsley. Comments on "modeling TCP reno performance: A simple model and its empirical validation". *IEEE/ACM Transactions on Networking*, 14(2), pp. 451–453, 2006.

[48] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17(1), pp. 1–14, 1989.

[49] H. K. J. Chu. Tuning TCP for the 21st century. Presentation at 75th IETF meeting, 2009.

[50] D. D. Clark. The design philosophy of the DARPA internet protocols. *ACM SIGCOMM Computer Communication Review*, 18(4), pp. 106–114, 1988.

[51] D. Clark, K. Sollins, J. Wrolawski, D. Katabi, J. Kulik, X. Yang, R. Braden, T. Faber, A. Falk, V. Pingali, M. Handley, and N. Chiappa. New Arch: Future generation Internet architecture. Final report of the DARPA NewArch project, 2003.

[52] A. Dan, H. Ludwig, and G. Pacifici. Web services differentiation with service level agreements. White paper, IBM Corporation, 2003.

[53] D. W. Davies. The control of congestion in packet-switching networks. *IEEE Transactions on Communications*, 20(3), pp. 546–550, 1972.

[54] J. D. Day. *Patterns in network architecture: a return to fundamentals*. Pearson Education, 2008.

[55] A. B. Downey. Lognormal and Pareto distributions in the Internet. *Computer Communications*, 28(7), pp. 790–801, 2005.

[56] N. Dukkipati and N. McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1), pp. 59–62, 2006.

[57] N. Dukkipati. *Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, 2007.

[58] N. Dukkipati, G. Gibb, N. McKeown, and J. Zhu. Building a RCP (Rate Control Protocol) test network. In *Proc. IEEE Symposium on High-Speed Interconnects*, 2007.

[59] R. Dunaytsev, Y. Koucheryavy, and J. Harju. The PFTK-model revised. *Computer Communications*, 29(13-14), pp. 2671–2679, 2006.

[60] C. Efstratiou, K. Cheverst, N. Davies, and A. Friday. An architecture for the effective support of adaptive context-aware applications. In *Proc. Conference on Mobile Data Management (MDM), LNCS 1987, Springer-Verlag*, pp. 15–26, 2001.

[61] L. Eggert and W. M. Eddy. Towards more expressive transport-layer interfaces. In *Proc. ACM/IEEE International Workshop on Mobility in the Evolving Internet Architecture*, pp. 71–74, 2006.

[62] M. A. El-Gendy, A. Bose, and K. G. Shin. Evolution of the Internet QoS and support for soft real-time applications. *Proceedings of the IEEE*, 91(7), pp. 1086–1104, 2003.

[63] G. Fairhurst and A. Sathiaseelan. Quick-Start for Datagram Congestion Control Protocol (DCCP). IETF Internet Draft, work in progress, June 2009. Published as RFC 5634.

[64] A. Falk, Y. Pryadkin, and D. Katabi. Specification for the explicit control protocol (XCP). IETF Internet Draft, work in progress, July 2007.

[65] J. Färber. *Modellierung von IP-basiertem Paketverkehr ausgewählter interaktiver Dienste*. PhD thesis (in German), University of Stuttgart, 2007.

[66] A. Feldmann, A. C. Gilbert, W. Willinger, and T. G. Kurtz. The changing nature of network traffic: Scaling phenomena. *ACM SIGCOMM Computer Communication Review*, 28(2), pp. 5–29, 1998.

[67] A. Feldmann. Internet clean-slate design: What and why? *ACM SIGCOMM Computer Communication Review*, 37(3), pp. 59–64, 2007.

[68] W. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, 10(4), pp. 513–528, 2002.

[69] M. Fisk and W. Feng. Dynamic right-sizing in TCP. In *2nd Annual Los Alamos Computer Science Institute Symposium (LACSI 2001)*, 2001.

[70] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4), pp. 392–403, 2001.

[71] B. Ford. Structured streams: a new transport abstraction. *ACM SIGCOMM Computer Communication Review*, 37(4), pp. 361–372, 2007.

[72] B. Ford and J. Iyengar. Breaking up the transport logjam. In *ACM Workshop on Hot Topics in Networks (HotNets-VII)*, 2008.

[73] S. B. Fredj, S. Oueslati-Boulahia, and J. W. Roberts. Measurement-based admission control for elastic traffic. In *Proc. 17th International Teletraffic Congress (ITC)*, pp. 161–172, 2001.

[74] X. Fu, H. Schulzrinne, A. Bader, D. Hogrefe, C. Kappler, G. Karagiannis, H. Tschofenig, and S. Van den Bosch. NSIS: A new extensible IP signaling protocol suite. *IEEE Communications Magazine*, 43(10), pp. 133–141, 2005.

[75] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *ACM SIGCOMM Computer Communication Review*, 39(1), pp. 68–73, 2009.

[76] L. A. Grieco and S. Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM SIGCOMM Computer Communication Review*, 34(2), pp. 25–38, 2004.

[77] K.-J. Grinnemo, J. Garcia, and A. Brunstrom. Taxonomy and survey of retransmission-based partially reliable transport protocols. *Computer Communications*, 27(15), pp. 1441–1452, 2004.

[78] M. Grossglauser and D. N. C. Tse. A time-scale decomposition approach to measurement-based admission control. *IEEE/ACM Transactions on Networking*, 11(4), pp. 550–563, 2003.

[79] L. Guo and I. Matta. The war between mice and elephants. In *Proc. International Conference on Network Protocols (ICNP)*, pp. 180–188, 2001.

[80] S. Ha, L. Le, I. Rhee, and L. Xu. Impact of background traffic on performance of high-speed TCP variant protocols. *Computer Networks*, 51(7), pp. 1748–1762, 2007.

[81] S. Ha and I. Rhee. Hybrid slow start for high-bandwidth and long-distance networks. In *Proc. PFLDnet2008*, March 2008.

[82] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating System Review*, 42(5), pp. 64–74, 2008.

[83] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2), pp. 207–233, 2003.

[84] S. Hauger, M. Scharf, J. Kögel, and C. Suriyajan. Quick-Start and XCP on a network processor: Implementation issues and performance evaluation. In *Proc. IEEE HPSR 2008*, 2008.

[85] J. Hautakorpi, G. Camarillo, R. Penfield, A. Hawrylyshen, and M. Bhatia. Requirements from SIP (Session Initiation Protocol) Session Border Control Deployments. IETF Internet Draft, work in progress, January 2009. Published as RFC 5853.

[86] B. R. Haverkort. *Performance of Computer Communication Systems – A Model-Based Approach*. Wiley, 1998.

[87] E. He, P. V.-B. Primet, M. Welzl, M. Goutelle, Y. Gu, S. Hegde, R. Kettimuthu, J. Leigh, C. Xiong, and M. M. Yousaf. A survey of transport protocols other than standard TCP. Document gfd-i.055, Global Grid Forum, 2005.

[88] J. Heidemann, K. Obraczkad, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5), pp. 616–630, 1997.

[89] J. Heidemann, K. Mills, and S. Kumar. Expanding confidence in network simulations. *IEEE Network*, 15(5), pp. 58–63, 2001.

[90] S. Hemminger. Network emulation with NetEm. In *Proc. Australia's National Linux Conference (LCA)*, 2005.

[91] S. Hessler and M. Welzl. An empirical study of the congestion response of RealPlayer, Windows MediaPlayer and Quicktime. In *Proc. IEEE International Symposium on Computers and Communications*, 2005.

[92] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. *ACM SIGCOMM Computer Communication Review*, 26(4), pp. 270–280, 1996.

[93] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. ACM MobiCom*, pp. 83–94, 2002.

[94] N. Hu and P. Steenkiste. Improving TCP startup performance using active measurements: algorithm and evaluation. *Proc. IEEE ICNP*, pp. 107–118, 2003.

[95] Communication Networks II lecture script. Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2008.

[96] S. Iren, P. D. Amer, and P. T. Conrad. The transport layer: Tutorial and survey. *ACM Computing Surveys*, 31(4), pp. 360–404, 1999.

[97] P. Jacobs and B. Davie. Technical challenges in the delivery of interprovider QoS. *IEEE Communications Magazine*, 43(6), pp. 112–118, 2005.

[98] V. Jacobson. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, 18(4), pp. 314–329, 1988.

[99] R. Jain. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications*, 4(7), 1986.

[100] R. Jain and K. K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer: Concepts, goals and methodology. In *Proc. Computer Networking Symposium*, pp. 134–143, 1988.

[101] M. Jain and C. Dovrolis. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proc. ACM Internet Measurement Conference (IMC)*, pp. 272–277, 2004.

[102] S. Jain, Y. Zhang, and D. Loguinov. Towards experimental evaluation of explicit congestion control. *Computer Networks*, 53(7), pp. 1027–1039, 2009.

[103] S. Jansen and A. McGregor. Simulation with real world network stacks. In *Proc. Winter Simulation Conference*, pp. 2454–2463, 2005.

[104] S. Jansen and A. McGregor. Performance, validation and testing with the network simulation cradle. In *Proc. IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, pp. 355–362, 2006.

[105] S. Jansen and A. McGregor. Static virtualization of C source code. *Software – Practice and Expererience*, 38(4), pp. 397–416, 2008.

[106] H. Jiang and C. Dovrolis. Passive estimation of TCP round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3), pp. 75–88, 2002.

[107] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions in Parallel and Distributed Systems*, 11(6), pp. 589–603, 2000.

[108] Jupiter Research. Retail Web site performance: Consumer reaction to a poor online shopping experience. Technical report, Jupiter Research, New York, NY, USA, 2006.

[109] M. Karsten. Collected experience from implementing RSVP. *IEEE/ACM Transactions on Networking*, 14(4), pp. 767–778, 2006.

[110] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM Computer Communication Review*, 32(4), pp. 89–102, 2002.

[111] F. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8(1), pp. 33–37, 1997.

[112] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49, pp. 237–252, 1998.

[113] F. Kelly. Fairness and stability of end-to-end congestion control. *European Journal of Control*, 9, pp. 159–176, 2003.

[114] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2), pp. 83–91, 2003.

[115] S. Keshav. *Congestion Control in Computer Networks*. PhD thesis, University of California, Berkeley, 1991.

[116] S. Kiesel and M. Scharf. Modeling and performance evaluation of transport protocols for firewall control. *Computer Networks*, 51(11), pp. 3232–3251, 2007.

[117] S. Kiesel. *Architekturen verteilter Firewalls für IP-Telefonie-Plattformen*. PhD thesis (in German), University of Stuttgart, 2008.

[118] C. C. Knestrick. Lunar: A user-level stack library for network emulation. Master's thesis, Virginia Polytechnic Institute and State University, 2004.

[119] S.-S. Ko and R. F. Serfozo. Response times in M/M/s fork-join networks. *Advances in Applied Probability*, 36(3), pp. 854–871, 2004.

[120] E. Kohler, S. Floyd, and A. Sathiaseelan. Faster restart for TCP friendly rate control (TFRC). IETF Internet Draft, work in progress, July 2008.

[121] K. Kumazoe, C. Marcondes, M. Gerla, D. Cavendish, M. Tsuru, and Y. Oi. Conservative slow start: Controlling losses in very high speed networks. In *Proc. IEEE ICC*, pp. 5798–5803, 2008.

[122] S. S. Kunniyur. AntiECN marking: A marking scheme for high bandwidth delay connections. In *Proc. IEEE International Conference on Communications (ICC)*, 2003.

[123] A. Kuzmanovic and E. W. Knightly. TCP-LP: low-priority service via end-point congestion control. *IEEE/ACM Transactions on Networking*, 14(4), pp. 739–752, 2006.

[124] A. Lakshmikantha, R. Srikant, N. Dukkipati, N. McKeown, and C. Beck. Buffer sizing results for RCP congestion control under connection arrivals and departures. *ACM SIGCOMM Computer Communication Review*, 39(1), pp. 5–15, 2009.

[125] R. Lange, N. Cipriani, L. Geiger, M. Großmann, H. Weinschrott, A. Brodt, M. Wieland, S. Rizou, and K. Rothermel. Making the World Wide Space happen: New challenges for the Nexus context platform. In *Proc. IEEE PerCom*, 2009.

[126] G. Lawton. New ways to build rich Internet applications. *Computer*, 41(8), pp. 10–12, 2008.

[127] L. Le, J. Aikat, K. Jeffay, and F. D. Smith. The effects of active queue management and explicit congestion notification on web performance. *IEEE/ACM Transactions on Networking*, 15(6), pp. 1217–1230, 2007.

[128] N. Leavitt. Browsing the 3D Web. *Computer*, 39(9), pp. 18–21, 2006.

[129] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. S. Wolff. The past and future history of the Internet. *Communications of the ACM*, 40(2), pp. 102–108, 1997.

[130] D. J. Leith, R. N. Shorten, and G. McCullagh. Experimental evaluation of Cubic-TCP. In *Proc. PFLDnet*, 2007.

[131] D. J. Leith, L. L. H. Andrew, T. Quetchenbach, R. N. Shorten, and K. Lavi. Experimental evaluation of delay/loss-based TCP congestion control algorithms. In *Proc. PFLDnet*, 2008.

[132] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1), pp. 1–15, 1994.

[133] K.-C. Leung and V. O. K. Li. Transmission Control Protocol (TCP) in wireless networks: issues, approaches, and challenges. *IEEE Communications Surveys & Tutorials*, 8(4), pp. 64–79, 2006.

[134] Y.-T. Li, D. Leith, and R. N. Shorten. Experimental evaluation of TCP protocols for high-speed networks. *IEEE/ACM Transactions on Networking*, 15(5), pp. 1109–1122, 2007.

[135] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1), pp. 36–60, 2006.

[136] D. Liu, M. Allman, S. Jin, and L. Wang. Congestion control without a startup phase. In *Proc. PFLDnet*, 2007.

[137] S. Liu, T. Başar, and R. Srikant. TCP-Illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation*, 65(6-7), pp. 417–440, 2008.

[138] M. Lorang. *Verbindungsorientierte und verbindungslose Mechanismen eines skalierbaren Verkehrsmanagements für IP-Netze*. PhD thesis (in German), University of Stuttgart, 2005.

[139] S. H. Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions on Networking*, 11(4), pp. 525–536, 2003.

[140] H.-L. Lu and I. Faynberg. An architectural framework for support of Quality of Service in packet networks. *IEEE Communications Magazine*, 6(41), pp. 98–105, 2003.

[141] W. Y. Lum and F. C. M. Lau. User-centric content negotiation for effective adaptation service in mobile computing. *IEEE Transactions on Software Engineering*, 29(12), pp. 1100–1111, 2003.

[142] P. Mähönen, D. Trossen, D. Papadimitriou, G. Polyzos, and D. Kennedy. The future networked society. White paper from the EIFFEL Think-Tank, 2006.

[143] C. L. T. Man, G. Hasegawa, and M. Murata. ImTCP: TCP with an inline measurement mechanism for available bandwidth. *Computer Communications*, 29(10), pp. 1614–1626, 2006.

[144] J. S. Marcus and D. Elixmann. The future of IP interconnection: Technical, economic, and public policy aspects. Report of WIK-Consult, Bad Honneff, Germany, 2008.

[145] R. Martin and M. Menth. Improving the timeliness of rate measurements. In *GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB)/Polish-German Teletraffic Symposium (PGTS)*, 2004.

[146] M. Mathies. Relentless congestion control. In *Proc. PFLDNeT*, 2009.

[147] M. Mathis, J. Semke, and J. Mahdavi. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3), pp. 67–82, 1997.

[148] S. McCanne and S. Floyd. ns network simulator. http://www.isi.edu/nsnam/ns/.

[149] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7), pp. 56–64, 2004.

[150] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *ACM SIGCOMM Computer Communication Review*, 35(2), pp. 37–52, 2005.

[151] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall PTR, 2nd edition, 2001.

[152] D. A. Menascé, V. A. F. Almeida, R. Riedi, F. Ribeiro, R. Fonseca, and W. Meira. A hierarchical and multiscale approach to analyze e-business workloads. *Performance Evaluation*, 54(1), pp. 33–57, 2003.

[153] A. Mielke. Elements for response-time statistics in ERP transaction systems. *Performance Evaluation*, 63(7), pp. 635–653, 2006.

[154] J. Milbrandt, M. Menth, and J. Junker. Experience-based admission control in the presence of traffic changes. *Journal of Communications (JCM)*, 2(1), pp. 10–21, 2007.

[155] R. B. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference, San Francisco, CA, USA*, pp. 267–277, 1968.

[156] P. Molinero-Fernández, N. McKeown, and H. Zhang. Is IP going to take over the world (of communications)? *ACM SIGCOMM Computer Communication Review*, 33(1), pp. 113–118, 2003.

[157] T. Moors. A critical review of "End-to-end arguments in system design". In *Proc. International Conference on Communications (ICC)*, pp. 1214–1219, 2002.

[158] P. Mosebekk. A Linux implementation and analysis of the eXplicit Control Protocol (XCP). Master's thesis, University of Oslo, 2005.

[159] A. P. Mudambi, X. Zheng, and M. Veeraraghavan. A transport protocol for dedicated end-to-end circuits. In *Proc. IEEE International Conference on Communications (ICC)*, volume 1, pp. 18–23, 2006.

[160] K. Nakauchi and K. Kobayashi. An explicit router feedback framework for high bandwidth-delay product networks. *Computer Networks*, 51(7), pp. 1833–1846, 2007.

[161] P. Natarajan, J. R. Iyengar, P. D. Amer, and R. Stewart. SCTP: An innovative transport layer protocol for the web. In *Proc. WWW 2006*, pp. 615–624, 2006.

[162] M. Necker, M. Scharf, and A. Weber. Performance of different proxy concepts in UMTS networks. In *LNCS 3427, Springer-Verlag*, pp. 36–51, 2005.

[163] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Transactions on Computers*, 37(6), pp. 739–743, 1988.

[164] J. Nichols, M. Claypool, and R. K. M. Li. Measurements of the congestion responsiveness of Windows streaming media. In *Proc. International ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pp. 94–99, 2004.

[165] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. *ACM SIGCOMM Computer Communication Review*, 27(4), pp. 155–166, 1997.

[166] A. Nurminen. Mobile, hardware-accelerated urban 3D maps in 3G networks. In *Proc. ACM Web3D*, pp. 7–16, 2007.

[167] A. Odlyzko. The delusions of net neutrality. In *Proc. Telecommunications Policy Research Conference*, 2008.

[168] J. Padhye, V. Firoiu, D. F. Towsley, and J. F. Kurose. Modeling TCP reno performance: A simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2), pp. 133–145, 2000.

[169] V. N. Padmanabhan and R. H. Katz. TCP fast start: A technique for speeding up Web transfers. In *Proc. IEEE Globecom*, pp. 41–46, 1998.

[170] G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1), pp. 101–106, 2006.

[171] R. Pan, B. Prabhakar, and K. Psounis. CHOKe - a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proc. IEEE INFOCOM*, volume 2, pp. 942–951, 2000.

[172] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe. Open research issues in Internet congestion control. IRTF Internet Draft, work in progress, August 2009. Published as RFC 6077.

[173] C. Partridge, D. Rockwell, M. Allman, R. Krishnan, and J. P. Sterbenz. A swifter start for TCP. Technical Report 8339, BBN Technologies, 2002.

[174] C. J. Pavlovski. Service delivery platforms in practice. *IEEE Communications Magazine*, 45(3), pp. 114–121, 2007.

[175] K. Pawlikowski, H.-D. J. Jeong, and J.-S. R. Lee. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, 40(1), pp. 132–139, 2002.

[176] V. Paxson and S. Floyd. Wide area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3), pp. 226–244, 1995.

[177] L. Peterson and V. S. Pai. Experience-driven experimental systems research. *Communications of the ACM*, 50(11), pp. 38–44, 2007.

[178] M. Poikselkä, G. Mayer, H. Khartabil, and A. Niemi. *The IMS: IP Multimedia Concepts and Services*. Wiley, 2nd edition, 2006.

[179] R. Prasad, C. Dovrolis, M. Murray, and kc claffy. Bandwidth estimation: Metrics, measurement techniques, and tools. *IEEE Network*, 17(6), pp. 27–35, 2003.

[180] M. Proebster. Leistungsuntersuchung von Verfahren zur Überlastregelung mit expliziter Router-Signalisierung. Diploma thesis (in German), Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2008.

[181] M. Proebster, M. Scharf, and S. Hauger. Performance comparison of router assisted congestion control protocols: XCP vs. RCP. In *Proc. 2nd International Workshop on the Evaluation of Quality of Service through Simulation in the Future Internet*, 2009.

[182] B. Raghavan and A. C. Snoeren. Decongestion control. In *ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-V)*, 2006.

[183] R. K. Reeser, R. D. van der Mei, and R. Hariharan. An analytic model of a Web server. In *Proc. 16th International Teletraffic Congress (ITC)*, pp. 1199–1208, 1999.

[184] I. Rhee and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. In *Proc. PFLDnet*, 2005.

[185] A. Riedl, M. Perske, T. Bauschert, and A. Probst. Investigation of the M/G/R processor sharing model for dimensioning of IP access networks with elastic traffic. In *Polish-German Teletraffic Symposium (PGTS)*, 2000.

[186] J. Roberts. Internet traffic, QoS and pricing. *Proceedings of the IEEE*, 92(9), pp. 1389–1399, 2004.

[187] L. G. Roberts. Major improvements in TCP performance over satellite and radio. In *Proc. IEEE MILCOM*, 2006.

[188] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), pp. 277–288, 1984.

[189] P. Sarolahti and A. Kuznetsov. Congestion control in Linux TCP. In *Proc. USENIX Annual Technical Conference*, pp. 49–62, 2002.

[190] P. Sarolahti, J. Korhonen, L. Daniel, and M. Kojo. Using Quick-Start to improve TCP performance with vertical hand-offs. In *Proc. IEEE Conference on Local Computer Networks*, pp. 897–904, 2006.

[191] P. Sarolahti. *TCP Performance in Heterogeneous Wireless Networks*. PhD thesis, University of Helsinki, 2007.

[192] P. Sarolahti, M. Allman, and S. Floyd. Determining an appropriate sending rate over an underutilized network path. *Computer Networks*, 51(7), pp. 1815–1832, 2007.

[193] A. Sathiaseelan and G. Fairhurst. Use of quickstart for improving the performance of TFRC-SP over satellite networks. In *Proc. International Workshop on Satellite and Space Communications*, 2006.

[194] M. Savorić. *Improving Congestion Control in IP-based Networks by Information Sharing*. PhD thesis, Technical University of Berlin, 2004.

[195] M. Scharf, M. C. Necker, and B. Gloss. The sensitivity of TCP to sudden delay variations in mobile networks. In *Proc. IFIP Networking 2004, LNCS 3042, Springer-Verlag*, pp. 76–87, 2004.

[196] M. Scharf. On the response time of large-scale composite Web services. In *Proc. 19th International Teletraffic Congress (ITC)*, pp. 1807–1816, 2005.

[197] M. Scharf and S. Kiesel. Head-of-line blocking in TCP and SCTP: Analysis and measurements. In *Proc. IEEE Globecom*, 2006.

[198] M. Scharf. Future Internet transport layer - heading towards a post-TCP era? Invited talk at Future Internet Design Workshop, European Conference on Optical Communications (ECOC), 2007.

[199] M. Scharf. Performance analysis of the Quick-Start TCP extension. In *Proc. IEEE Broadnets*, 2007.

[200] M. Scharf and C. Zeeh. Experience with simulating real TCP/IP-protocol stacks. Presentation at ITG Fachgruppe 5.2.1 Workshop on Network Engineering, 2007.

[201] M. Scharf. Why do we need TCP flow control (rwnd)? E-mail discussion on the IRTF end-to-end interest mailing list, June 2008.

[202] M. Scharf, S. Floyd, and P. Sarolathi. TCP flow control for fast startup schemes. IETF Internet Draft, work in progress, July 2008.

[203] M. Scharf, S. Hauger, and J. Kögel. Quick-Start TCP: From theory to practice. In *Proc. PFLDnet*, 2008.

[204] M. Scharf and H. Strotbek. Performance evaluation of Quick-Start TCP with a Linux kernel implementation. In *Proc. IFIP Networking 2008, LNCS 4982, Springer-Verlag*, pp. 703–714, 2008.

[205] M. Scharf. Performance evaluation of fast startup congestion control schemes. In *Proc. IFIP Networking 2009, LNCS 5550, Springer-Verlag*, pp. 716–727, 2009.

[206] M. Scharf. Some thoughts on Quick-Start TCP admission control algorithms. Presentation at IRTF ICCRG meeting, May 2009.

[207] M. Scharf, M. Eissele, C. Mueller, and T. Ertl. Speeding up the 3D Web: A case for fast startup congestion control. In *Proc. PFLDNeT*, 2009.

[208] T. R. Schmidt. *Adaptionsalgorithmen zur Erhöhung der Dienstgüte verteilter interaktiver Multimedia-Anwendungen in IP-basierten Netzen*. PhD thesis (in German), University of Stuttgart, 2004.

[209] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The new Web: Characterizing AJAX traffic. In *Proc. Passive and Active Network Measurement Conference, LNCS 4979, Springer-Verlag*, pp. 31–40, 2008.

[210] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP buffer tuning. *ACM SIGCOMM Computer Communication Review*, 28(4), pp. 315–323, 1998.

[211] S. Shalunov, L. D. Dunn, Y. Gu, S. Low, I. Rhee, S. Senger, B. Wydrowski, and L. Xu. Design space for a bulk transport tool. Bulk transport working group report, Internet2, 2005.

[212] S. Shenker. Fundamental design issues for the future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7), pp. 1176–1188, 1995.

[213] R. Shorten and D. Leith. H-TCP: TCP for high-speed and long-distance networks. In *Proc. PFLDnet*, 2004.

[214] A. Shriram and J. Kaur. Empirical evaluation of techniques for measuring available bandwidth. In *Proc. IEEE INFOCOM*, pp. 2162–2170, 2007.

[215] B. Sikdar, S. Kalyanaraman, and K. S. Vastola. Analytic models for the latency and steady-state throughput of TCP Tahoe, Reno, and SACK. *IEEE/ACM Transactions on Networking*, 11(6), pp. 959–971, 2003.

[216] M. Simeoni, P. Inverardi, A. D. Marco, and S. Balsamo. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5), pp. 295–310, 2004.

[217] L. Skorin-Kapov, M. Mosmondor, O. Dobrijevic, and M. Matijasevic. Application-level QoS negotiation and signaling for advanced multimedia services in the IMS. *IEEE Communications Magazine*, 45(7), pp. 108–116, 2007.

[218] M. Smith and S. Bishop. Flow control in the Linux network stack. Technical report, Computer Laboratory, University of Cambridge, 2005.

[219] M. Sridharan, D. Bansal, and D. Thaler. Implementation report on experiences with various TCP RFCs. Presentation at 68th IETF meeting, 2007.

[220] R. Srikant. *The mathematics of Internet congestion control.* Birkhäuser Boston, 2004.

[221] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. *ACM SIGCOMM Computer Communication Review*, 29(4), pp. 81–94, 1999.

[222] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Transactions on Networking*, 11(1), pp. 33–46, 2003.

[223] H. Strotbek. Entwurf und Implementierung einer TCP-Erweiterung im Linux-Kernel. Diploma thesis (in German), Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2007.

[224] A.-J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind Akamai (travelocity-based detouring). *ACM SIGCOMM Computer Communication Review*, 36(4), pp. 435–446, 2006.

[225] M. Suchara, R. Witt, and B. Wydrowski. TCP MaxNet - implementation and experiments on the WAN in Lab. In *Proc. IEEE International Conference on Networks (ICON)*, 2005.

[226] C. Suriyajan. Design and Implementation of Quick-Start and XCP Router Functions in a Network Processor. Master thesis, Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2007.

[227] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM*, 2006.

[228] A. Thomasian. Performance analysis of database systems. In *Performance Evaluation: Origins and Directions, LNCS 1769, Springer-Verlag*, pp. 305–327, 2000.

[229] M. Tian, A. Gramm, T. Naumowicz, and H. Ritter. A concept for QoS integration in Web services. In *Proc. Web Services Quality Workshop, Rome, Italy*, 2003.

[230] P. Tran-Gia, A. Feldmann, R. Steinmetz, J. Eberspächer, M. Zitterbart, P. Müller, and H. Schotten. G-Lab Phase 1. White paper (in German), German Lab, 2008.

[231] U. Vallamsetty, K. Kant, and P. Mohapatra. Characterization of e-commerce traffic. *Electronic Commerce Research*, 3(1-2), pp. 167–192, 2003.

[232] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 36(SI), pp. 329–343, 2002.

[233] C. Villamizar and C. Song. High performance TCP in ANSNET. *ACM SIGCOMM Computer Communication Review*, 24(5), pp. 45–60, 1994.

[234] A. Vishwanath, V. Sivaraman, and M. Thottan. Perspectives on router buffer sizing: recent results and open problems. *ACM SIGCOMM Computer Communication Review*, 39(2), pp. 34–39, 2009.

[235] V. Visweswaraiah and J. Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, University of Southern California, 1997.

[236] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded web servers. In *Proc. USENIX Annual Technical Conference, Boston, MA, USA*, pp. 189–202, 2001.

[237] S. Y. Wang, C. L. Chou, and C. Lin. The design and implementation of the NCTUns network simulation engine. *Simulation Modelling Practice and Theory*, 15, pp. 57–81, 2007.

[238] S. Wanke, M. Scharf, S. Kiesel, and S. Wahl. Measurement of the SIP parsing performance in the SIP Express Router. In *Proc. EUNICE 2007, LNCS 4606, Springer-Verlag*, pp. 103–110, 2007.

[239] Web site. WAN in Lab – Traffic Traces for TCP Evaluation. http://wil.cs.caltech.edu/suite/TrafficTraces.php.

[240] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler. *The Linux Networking Architecture*. Prentice Hall, 2005.

[241] D. X. Wei and P. Cao. NS-2 TCP-Linux: An NS-2 TCP implementation with congestion control algorithms from Linux. In *Proc. ValueTool'06 – Workshop of NS-2*, 2006.

[242] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: Motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6), pp. 1246–1259, 2006.

[243] J. Wei and C.-Z. Xu. eQoS: Provisioning of client-perceived end-to-end QoS guarantees in Web servers. *IEEE Transactions on Computers*, 55(12), pp. 1543–1556, 2006.

[244] M. C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, and F. D. Smith. Tmix: A tool for generating realistic TCP application workloads in ns-2. *ACM SIGCOMM Computer Communication Review*, 36(3), pp. 65–76, 2006.

[245] M. Welzl. Traceable congestion control. In *Proceedings of QoFIS/ICQT, LNCS 2511, Springer-Verlag*, pp. 273–282, 2002.

[246] M. Welzl. *Network congestion control: managing Internet traffic*. Wiley, 2005.

[247] M. Welzl and W. M. Eddy. Congestion control in the RFC series. IRTF Internet Draft, work in progress, October 2008. Published as RFC 5783.

[248] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network*, 15(3), pp. 28–37, 2001.

[249] S. Wright. Admission control in multi-service IP networks: A tutorial. *IEEE Communications Surveys & Tutorials*, 9(2), pp. 72–87, 2007.

[250] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One more bit is enough. *ACM SIGCOMM Computer Communication Review*, 35(4), pp. 37–48, 2005.

[251] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz. P4P: Provider portal for applications. *ACM SIGCOMM Computer Communication Review*, 38(4), pp. 351–362, 2008.

[252] N. Yin and M. G. Hluchyj. Implication of dropping packets from the front of a queue. *IEEE Transactions on Communications*, 41(6), pp. 846–851, 1993.

[253] C. Zeeh. Integration of the Linux TCP/IP Protocol Stack in an Event-driven Simulation Environment. Diploma thesis, Institute of Communication Networks and Computer Engineering, University of Stuttgart, 2006.

[254] Y. Zhang, L. Qiu, and S. Keshav. Speeding up short data transfers: Theory, architectural support, and simulation results. In *Proc. NOSSDAV*, 2000.

[255] Y. Zhang and T. R. Henderson. An implementation and experimental study of the Explicit Control Protocol (XCP). In *Proc. IEEE Infocom*, pp. 1037–1048, 2005.

[256] Y. Zhang, D. Leonard, and D. Loguinov. Jetmax: Scalable max-min congestion control for high-speed heterogeneous networks. *Computer Networks*, 52(6), pp. 1193–1219, 2008.

[ATM TM] ATM Forum. Traffic Management Specification, version 4.1. Technical Report AF-TM-0121.000, The ATM Forum, 1999.

[ES 282 001] ETSI. Telecommunications and Internet converged Services and Protocols for Advanced Networking (TISPAN); NGN Functional Architecture Release 1. ES 282 001, ETSI, 2008.

[802.1D] IEEE. Local and metropolitan area networks Media Access Control (MAC) Bridges. Standard 802.1D-2004, IEEE, 2004.

[1003.1] IEEE. Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. Standard 1003.1-2008, IEEE, 2008.

[RFC 791] J. Postel. Internet Protocol. RFC 791, IETF, September 1981.

[RFC 793] J. Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

[RFC 896] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, IETF, January 1984.

[RFC 1287] D. Clark, L. Chapin, V. Cerf, R. Braden, and R. Hobby. Towards the Future Internet Architecture. RFC 1287, IETF, December 1991.

[RFC 1323] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, IETF, May 1992.

[RFC 1633] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633, IETF, June 1994.

[RFC 1644] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC 1644, IETF, July 1994.

[RFC 1958] B. Carpenter (Editor). Architectural Principles of the Internet. RFC 1958, IETF, June 1996.

[RFC 2140] J. Touch. TCP Control Block Interdependence. RFC 2140, IETF, April 1997.

[RFC 2309] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, IETF, April 1998.

[RFC 2475] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475, IETF, December 1998.

[RFC 2581] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999.

[RFC 2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF, June 1999.

[RFC 2861] M. Handley, J. Padhye, and S. Floyd. TCP Congestion Window Validation. RFC 2861, IETF, June 2000.

[RFC 2914] S. Floyd. Congestion Control Principles. RFC 2914, IETF, September 2000.

[RFC 3124] H. Balakrishnan and S. Seshan. The Congestion Manager. RFC 3124, IETF, June 2001.

[RFC 3135] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, IETF, June 2001.

[RFC 3168] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, September 2001.

[RFC 3390] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, IETF, October 2002.

[RFC 3649] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, IETF, December 2003.

[RFC 3742] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742, IETF, March 2004.

[RFC 4094] J. Manner and X. Fu. Analysis of Existing Quality-of-Service Signaling Protocols. RFC 4094, IETF, May 2005.

[RFC 4594] J. Babiarz, K. Chan, and F. Baker. Configuration Guidelines for DiffServ Service Classes. RFC 4594, IETF, August 2006.

[RFC 4614] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614, IETF, September 2006.

[RFC 4782] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782, IETF, January 2007.

[RFC 5290] S. Floyd and M. Allman. Comments on the Usefulness of Simple Best-Effort Traffic. RFC 5290, IETF, July 2008.

[RFC 5348] S. Floyd, M. Handley, J. Padhye, and J. Widmer. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 5348, IETF, September 2008.

[RFC 5559] P. Eardley. Pre-Congestion Notification (PCN) Architecture. RFC 5559, IETF, June 2009.

[ISO 7498] ISO. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. International Standard 7498-1, ISO/IEC, 1994.

[E.800] ITU-T. Terms and definitions related to quality of service and network performance including dependability. Rec. E.800, ITU-T, 1994.

[G.1000] ITU-T. Communications Quality of Service: A framework and definitions. Rec. G.1000, ITU-T, 2001.

[G.1010] ITU-T. End-user multimedia QoS categories. Rec. G.1010, ITU-T, 2001.

[G.1040] ITU-T. Network contribution to transaction time. Rec. G.1040, ITU-T, 2006.

[Y.1221] ITU-T. Traffic control and congestion control in IP-based networks. Rec. Y.1221, ITU-T, 2002.

[Y.1291] ITU-T. An architectural framework for support of Quality of Service in packet networks. Rec. Y.1291, ITU-T, 2004.

[Y.1541] ITU-T. Network performance objectives for IP-based services. Rec. Y.1541, ITU-T, 2002.

[Y.2001] ITU-T. General overview of NGN. Rec. Y.2001, ITU-T, 2004.

[Y.2011] ITU-T. General principles and general reference model for Next Generation Networks. Rec. Y.2011, ITU-T, 2004.

[TS 23.107] 3GPP. Quality of Service (QoS) concept and architecture. TS 23.107, 3GPP, 2007.

[TS 23.228] 3GPP. IP Multimedia Subsystem (IMS); stage 2. TS 23.228, 3GPP, 2008.