Experiences with Implementing Quick-Start in the Linux Kernel

Michael Scharf <michael.scharf@ikr.uni-stuttgart.de>
Haiko Strotbek <haiko@strotbek.com>

University of Stuttgart, Germany

IETF 69 - TSVAREA - July 24, 2007

This work is partly funded by the German Research Foundation (DFG) through the Center of Excellence (SFB) 627 "Nexus".

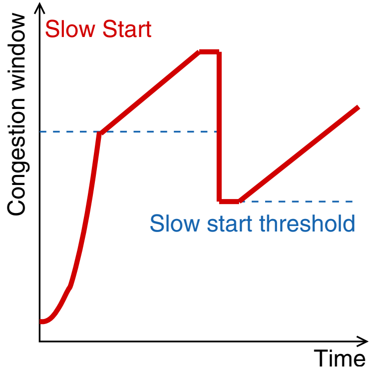
Agenda

1. **Overview of Quick-Start**
2. **Implementation in the Linux kernel**
3. **Initial measurement results**
4. **Lessons learnt**
5. **Conclusions and future work**

Overview of Quick-Start

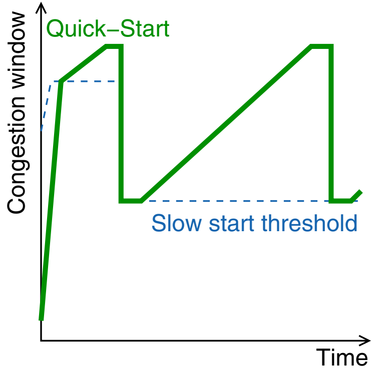
Slow-Start in TCP (RFC 2581)

- **Exponential growth of congestion window**
  - After connection setup or long idle periods
  - (After retransmission timeouts)
- **One pillar of TCP congestion control**
  - Probing of path capacity
  - Initialization of ACK clocking mechanism



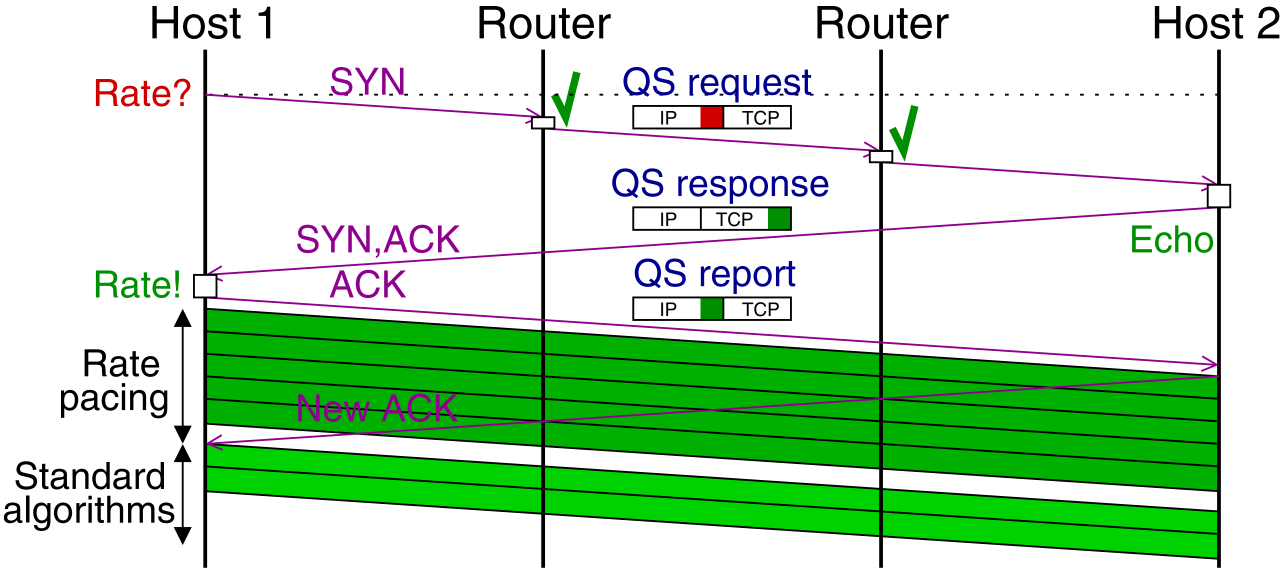
Quick-Start in TCP (RFC 4782)

- **Idea: Start immediately with high sending rate**
  - Reduces delay for interactive applications
  - Requires explicit feedback from routers on path
- **Experimental RFC since Jan. 2007**



Overview of Quick-Start

Example: Quick-Start During 3-Way Handshake



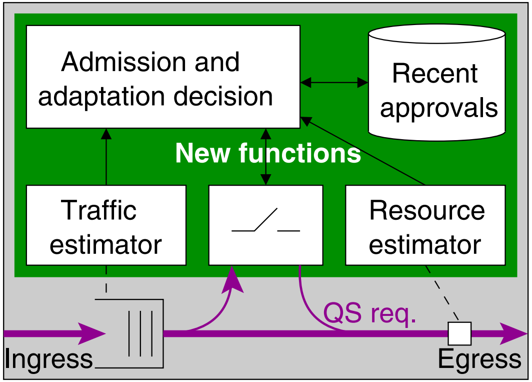
- **IP and TCP options to "request" for a data rate (no QoS reservation!)**
- **Raw granularity: 15 steps from 80 kbit/s to 1.31 Gbit/s**
- **All routers on path must explicitly approve request**

Overview of Quick-Start

Additional Router Functions

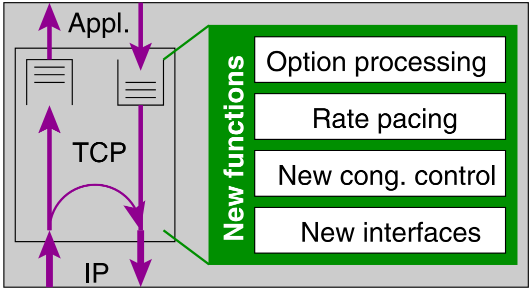
- **Processing of new IP options**
- **Avail. bandwidth on egress interfaces**
  - Link capacity (cross-layer issue!)
  - Traffic metering
- **Admission control of QS requests**

➥ **No per-flow state**



Additional Host Functions

- **Read/write new IP and TCP options**
- **Modified congestion and flow control**
  - Rate pacing after QS approvals
  - Additional state/information storage



Overview of Quick-Start

Benefits of Quick-Start

- **Speeds up interactive applications when RTT is large**
  - After connection setup
  - After longer idle period (with cong. window validation acc. to RFC 2861)
- **Could complement other new high-speed TCP extensions**
- **Conservative alternative to non standard-compliant workarounds**

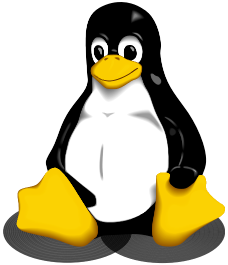
Challenges

- ***Deployment*: Requires support by all routers (and middleboxes)**
- ***Cross-layer issues*: Routers have to estimate available bandwidth**
- ***Security*: Can be rendered useless by attackers**
- ***Real-world experience*: Mostly studied by simulations so far**

➥ **Recommended for controlled environments only, *not the Internet***

Implementation in the Linux Kernel

Our Quick-Start Implementation in Linux



- **(Almost) all host and router functions**
- **Modified kernel (currently based on 2.6.20.11)**
- **TCP and IPv4 only**

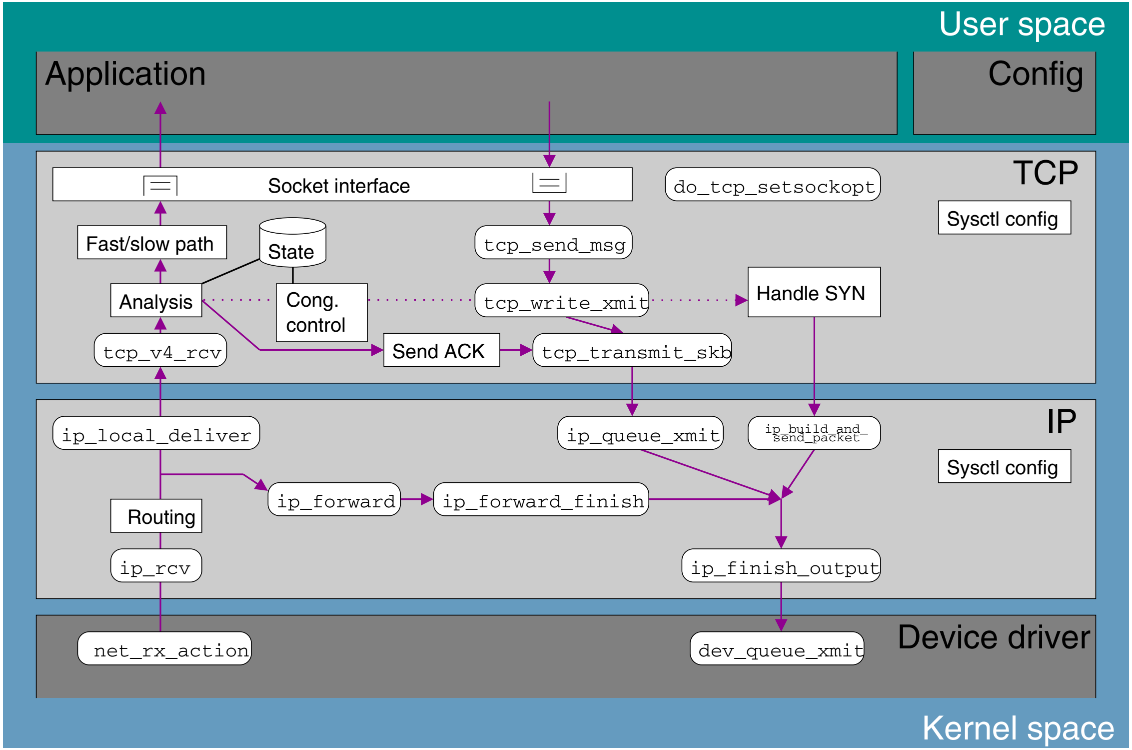
➥ **Works in lab tests correctly**

Some Statistics

- **Limited Effort**
  - 1700 lines of code (5 person months)
  - Changes affect 20 different files
- **Additional state information**
  - Host: About 20 integer variables per TCP connection
  - Router: Some integer variables per egress interface
- **Configuration: >10 new sysctl options**

Implementation in the Linux Kernel

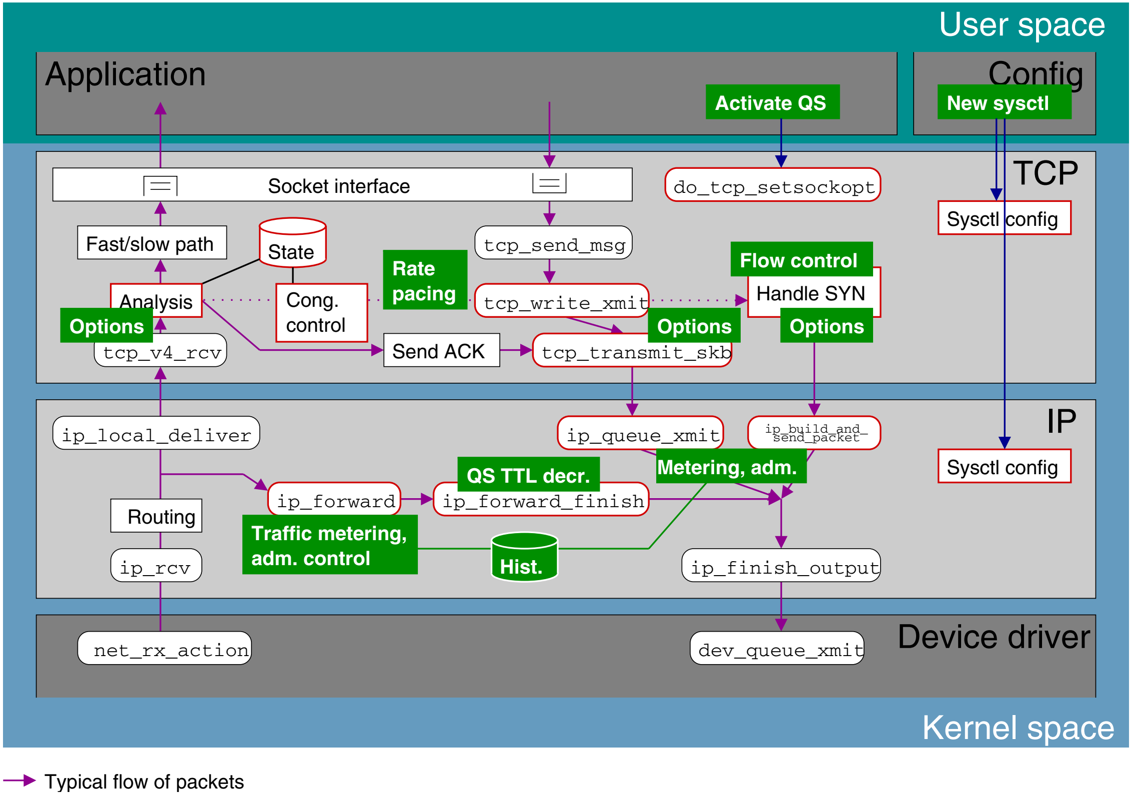
Linux Stack (Simplified)



→ Typical flow of packets

Implementation in the Linux Kernel

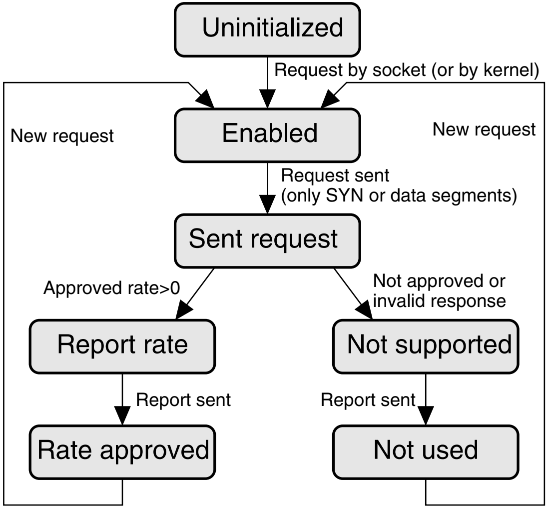
Linux Stack (Simplified) - Code Modifications



Implementation in the Linux Kernel

Sender State Engine

- **Diabled by default**
- **Enabling of Quick-Start**
    - By application via socket option
    - By heuristics inside kernel
- **Reasons for further states**
    - Requests only in SYN or data segments
    - Sending rate reports



Uninitialized

Request by socket (or by kernel)

New request

Enabled

New request

Request sent
(only SYN or data segments)

Sent request

Approved rate>0

Not approved or
invalid response

Report rate

Not supported

Report sent

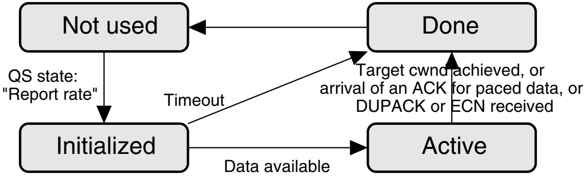
Report sent

Rate approved

Not used

Rate Pacing States

- **Rate pacing starts when first data segment is sent, not earlier**
- **Several abort conditions**



Not used

Done

QS state:
"Report rate"

Timeout

Target cwnd achieved, or
arrival of an ACK for paced data, or
DUPACK or ECN received

Initialized

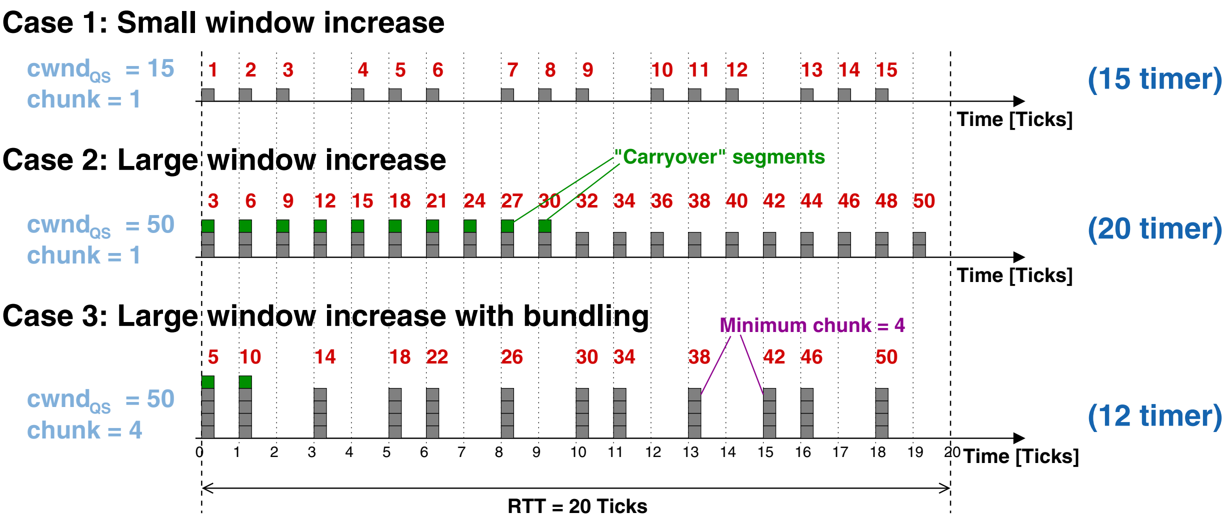
Active

Data available

Implementation in the Linux Kernel

Rate Pacing - Realization Details

- **Usage of internal kernel timers**
  - Linux kernel has a high timer granularity (up to 1000 Hz)
  - Limitation of the number of timers by "minimum chunk size" parameter
- **Timer initialization has to handle different cases**



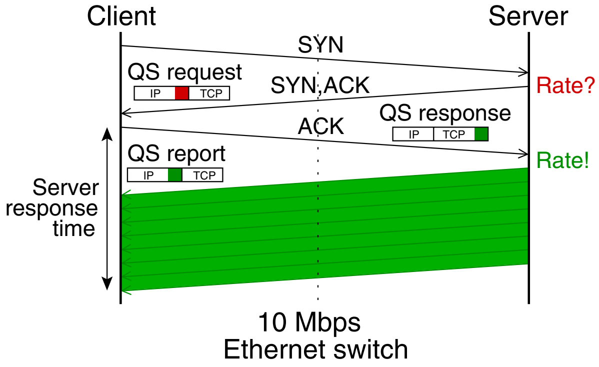
Initial Measurement Results

Processing Overhead (CPU Effort)

- **Hosts function (TCP layer): No additional CPU load measured so far**
  - Rate pacing rather lightweight
  - In total, only small parts of TCP code modified
- **Router function (IP layer): CPU load increase observed (ca. +15%)**
  - Reason: Per-packet processing for traffic metering
  - No significant impact of Quick-Start specific functions

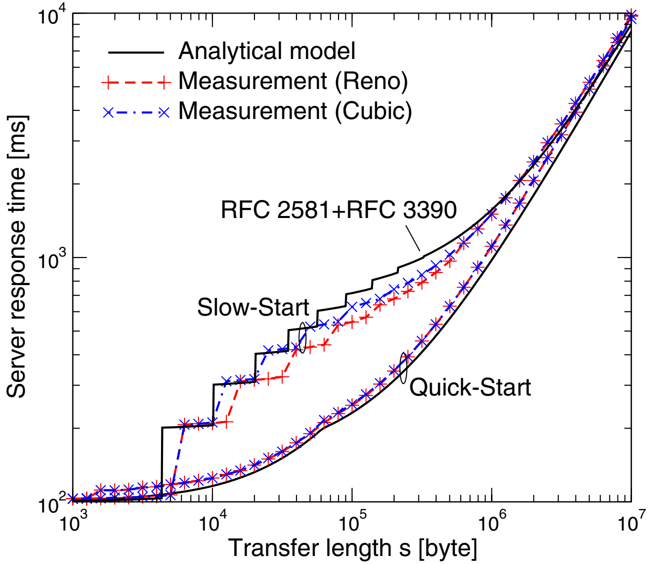
TCP Performance Benefit

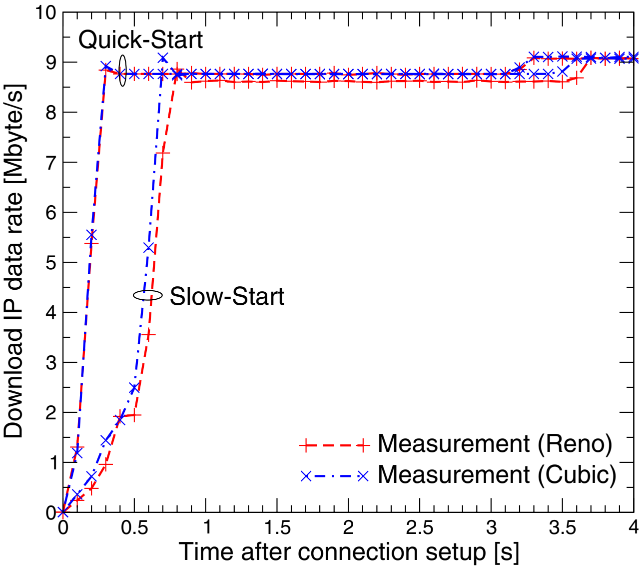
- **Transfer time reduction depends on bandwidth-delay product (BDP)**
- **Testbed example**
  - 10Mbps Ethernet
  - 100ms RTT (realized by netem)
  - ➥ BDP of 84 segments



Initial Measurement Results

Example: Improvement of Transfer Times





- **Details of analytical analysis: Michael Scharf, "Performance Analysis of the Quick-Start TCP Extension", Proc. IEEE Broadnets, Sept. 2007**
- **Work in progress: Measurements with real applications**

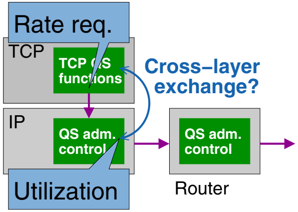
Lessons Learnt

Observations (1)

- **Interaction with flow control: Automatic buffer tuning announces too small receive windows, and interaction with window TCP scale option**
  - ➥ see draft-scharf-tsvwg-quick-start-flow-control-01.txt
- **TCP and IP option handling is tricky in practice**
  - Options are processed at several different places in the stack
  - Setting IP options from TCP code is not forseen by the standard APIs
  - TCP MSS must be reduced to leave space for IP options
  - ➥ Requires several workarounds and expanded APIs
- **Drivers do not reliably tell link capacity (interface card speed)**
  - Current solution: Manual configuration per sysctl interface
  - Potential alternative: Active bandwidth probing (?)
- **Mid-connection usage less straightforward than connection setup**

Lessons Learnt

Observations (2)



- **TCP connection end-points must have QS router processing**
  ➥ Potential for cross-layer information exchange
- **Setting of `ssthresh` after Quick-Start approval is an important design choice. Current solution:**
  - If QS request is reduced by routers: $\texttt{ssthresh = cwnd}_{\texttt{QS}}$
  - If QS request is not reduced: $\texttt{ssthresh = 2*cwnd}_{\texttt{QS}}$

  ➥ Is not optimal in all cases!

Open Issues

- **Interaction of rate pacing and Nagle algorithm**
- **IPv6**
- **Path MTU discovery**
- **Automatic self-configuration (reduction of number of parameters)**

Conclusions and Future Work

Conclusions

- **We do have running code ;)**
- **Not too difficult to implement Quick-Start in the Linux stack**
  - Overall implementation straightforward
  - But: Small modifications at many places, some ugly workarounds
- **No major issues found in spec, except for flow control interaction (see draft-scharf-tsvwg-quick-start-flow-control-01.txt)**
- **Still, any explicit router feedback is challenging...**

Ongoing/Future Work

- **Make kernel patch available to allow real-world performance tests**
- **Quick-Start router functions in a network processor**
  - Intel IXP2400
  - Ongoing work at the University of Stuttgart