Copyright Notice

Institute of Communication Networks and Computer Engineering
University of Stuttgart
Pfaffenwaldring 47, 70569 Stuttgart, Germany
Phone: ++49-711-685-68026, Fax: ++49-711-685-67983
email: mail@ikr.uni-stuttgart.de, http://www.ikr.uni-stuttgart.de

Scheduling Algorithms for Simultaneous Software Updates of Electronic Devices in Vehicles

Jörg Sommer[1], Volker Feil[2], and Enrique Adeva Sanz[2]

[1]University of Stuttgart, Institute of Communication Networks and Computer Engineering (IKR)
Pfaffenwaldring 47, 70569 Stuttgart, Germany
Email: joerg.sommer@ikr.uni-stuttgart.de

[2]Daimler AG, Infotainment and Telematics
HPC 050/G021, 71059 Sindelfingen, Germany
Email: volker.feil@daimler.com

***Abstract*— Today's upper-class passenger cars have various interconnected electronic devices. Each device performs complex functions, enabled by software that can be stored in a flash memory. Of these, the devices in the multimedia and infotainment domain contain by far the most software with a size in the order of one Gbyte. In this domain, the devices are the performance bottlenecks, not the communication systems. Throughout the vehicle life cycle, parts of the software have to be frequently updated during maintenance. Today, the software of the devices is updated in a consecutive manner. Due to performance bottlenecks caused by the affected devices, the update can take a long time that leads to high costs.**

Therefore, the objective is to reduce the total update time by a higher utilization of the common bus resource. In this paper, we introduce and investigate algorithms that update the software of multiple devices simultaneously and evaluate the efficiency of these algorithms. We focus on scheduling algorithms on the Application layer and the *Logical Link Control* (LLC) layer and model the update process by means of Petri nets. Our studies show that it is most promising to combine a simple scheduling algorithm on the Application layer with Round Robin on the LLC layer.

I. Introduction

The number of interconnected electronic devices in today's upper-class passenger cars, also called *Electronic Control Units*, has tremendously increased to more than 70. Most of the devices have an inner flash memory that stores software. Nowadays, most innovations in automotive systems are directly or indirectly enabled by software [1].

The observational Moore's law states that the number of transistors that can be inexpensively placed on an integrated circuit is increasing exponentially. As we notice this law is also valid for sizes of flash memories. Therefore, the amount of in-vehicle software stored in flash memories is increasing exponentially, too.

Within a vehicle, the multimedia and infotainment domain contains by far the most software. The actual software size depends on the installed equipment. For instance, a full equipped multimedia and infotainment domain including navigation map data is of the order of one Gbyte.

During maintenance, often parts of the vehicular software have to be updated. The large amount of data results in a long maintenance time, which in turn leads to high costs. For this reason, the software update time has to be reduced. Especially in the multimedia and infotainment domain, it is a challenge to update the software in an acceptable time. In this paper, we analyze algorithms that reduce the update time by simultaneous flash programming of several electronic devices those are interconnected by an in-vehicle bus system. So far, the updating of flash memories of several devices is done consecutively.

The update software is transferred from a source to a target device via networks that are mainly used for exchanging data of normal operation. In many cases, the source software is located on a device outside the car. For instance, during maintenance this software is mainly located on a so-called *diagnostic tester* in the workshop. It is also possible that the software source is located closer to the target device. Particularly in the multimedia and infotainment domain, this solution is often applied due to the need to provide a high bandwidth channel to the target devices. The current Mercedes-Benz solution is to insert a *Compact Disc* (CD) that contains the appropriate software into the CD player in the multimedia and infotainment domain. All devices within this domain are connected to this CD player via a high-bandwidth bus system that is called *Media Oriented Systems Transport* (MOST).

Usually, the target devices are the performance bottlenecks and not the bus system. The first reason for a target device's performance limitation is the slowness of receiving data from the bus system. In the case of MOST, the performance of the connection between the *Extended Host Controller* (EHC) and the *Network Interface Controller* (NIC) depends generally on the requirements for normal operation. For instance, if a mobile phone interface device has to receive only control commands like *DIAL_NUMBER* or *INCOMING_CALL*, then the receiver's connection is realized cost-efficient. Therefore, the connection is narrowband in spite of the used high bandwidth bus system MOST. Anyway, the actual benefit is the possibility to transmit the speech audio streams from the mobile phone interface device to an audio amplifier via MOST. Updating the software of the mobile phone interface device

via the narrowband receiver connection leads certainly to a low utilization of MOST, if there are no other additional data exchange operations.

The second reason for the performance limitation of a target device is the slowness of flash memory deletion and writing. The writing rates of flash memories used in the multimedia and infotainment domain are of the order of 500 kbps. However, the maximum data rate of flash memories does not in general limit the writing performance. By using flash memories in parallel, the writing rates can be increased.

The basic motivation for updating multiple devices simultaneously instead of consecutively is to reduce the total update time by a higher utilization of the common bus resource. The remainder of this paper investigates algorithms that allow simultaneous updating by scheduling partial activities of single software updates. The next section describes the relevant technologies, protocols, and processes that are needed for the software update. Section III presents the modeling of the update process by means of Petri nets. In section IV the relevant scheduling algorithms are explained and are evaluated in section V. Finally, a conclusion and outlook is given in section VI.

II. Technical Bases

Currently, the protocol stack shown in Figure 1 could be used in passenger cars for updating software in the multimedia and infotainment domain, e.g., in Mercedes-Benz cars. The protocol stack comprises four major layers: MOST on the Physical layer and the *Medium Access Control* (MAC) layer, the *MOST High Protocol* on the *Logical Link Control* (LLC) layer, and the *Unified Diagnostic Services* (UDS) on the Application layer. Figure 1 shows the encapsulation of protocol data units, too. In the following subsections, the features and mechanisms of the used protocols are described in more detail.

A. MOST

Media Oriented Systems Transport (MOST) [2], defined in 1998 by a consortium of automobile manufactures and component suppliers, is a serial communication system intended for transmitting audio and video data, burst-like data, and control data via, e.g., polymeric optical fibers. Particularly in passenger cars, MOST is used to realize multimedia and infotainment systems composed of, e.g., a CD-/DVD-player, a TV receiver, a navigation system, a mobile phone interface device, an iPod interface device, and an audio amplifier in a modular manner. MOST supports three types of channels: synchronous, asynchronous, and control channel. The synchronous channel is used for transferring time-sensitive streaming data. The asynchronous channel is mainly used for transporting burst data traffic (e.g., caused by Internet applications). It can be used for the software update, too. The control channel is used for exchanging control messages with low bandwidth requirements. A MOST system has a timing master that generates frames. The frame rate is usually 44.1 kHz. Per frame, 64 bytes are subdivided into the three channels synchronous channel, asynchronous channel, control channel, and administrative tasks. Therefore, MOST has a data rate equal to $44.1\ \text{kHz} * 64 * 8\ \text{bit} \approx 22.6\ \text{Mbps}$. However, the asynchronous channel can use in maximum 36 bytes per frame. The corresponding data rate is equal to $44.1\ \text{kHz} * 36 * 8\ \text{bit} \approx 12.7\ \text{Mbps}$.

The presented MOST is a widespread, well-established multimedia networking technology. Currently, there are efforts to double (or even to increase by factor six) the bandwidth by introducing appropriate technology extensions. A more detailed description about MOST can be found in [2], [3].

B. MOST High Protocol

The MOST High Protocol (MHP) [4] is a reliable communication protocol. In general, MHP is applied for updating the software via an asynchronous channel. At first, a connection between sender and receiver has to be established. Afterwards, the sender can transmit MHP packets to the receiver. Either single MHP packets or blocks that consist of multiple MHP packets are acknowledged after their reception. Therefore, the receiver transmits an acknowledge packet to the sender. Finally, the connection has to be released. The maximum length of a packet transmitted over the asynchronous channel is 1024 bytes. For transferring software update data only 1006 bytes can be used for the payload. The protocol information contained in the header and trailer has always a length of 18 bytes. The transmission of a MHP packet that has a length of 1024 bytes needs $\lceil 1024/36 \rceil = 29$ MOST frames. Therefore, the data rate for the payload decreases to $(44.1/29\ \text{kHz}) * 1006 * 8\ \text{bit} \approx 12.2\ \text{Mbps}$.

Furthermore, the data rate degrades due to MOST's arbitration mechanism. A sender has to wait at least four MOST frames before sending a packet again. Therefore, the nominal data rate is equal to $(44.1/(29+4)\ \text{kHz}) * 1006 * 8\ \text{bit} \approx 10.8\ \text{Mbps}$.

The MHP connection handling does not influence this nominal data rate if the connection is established for an appropriate long time.

C. Unified Diagnostic Services

Unified Diagnostic Services (UDS) has been standardized by the ISO [5]. This standard defines diagnostic services for road vehicles including passenger cars. UDS is a protocol on the Application layer.

A target device inside a vehicle can offer these services, and a diagnostic tester, e.g., in the workshop, can call these services. A service call is realized by a service request message sent from diagnostic tester to a target device. After service execution, the target device has to respond with a service response message. UDS defines this diagnostic service request/response protocol as well as a diagnostic services set.

For software updates the target devices generally have to offer a so-called *Transfer Data Service*. By this service, the software update data can be transferred to the target device. Each update software is segmented, and each segment is sent in the payload of a *Transfer Data Request* message. After receipt of this message, the target device writes the segment

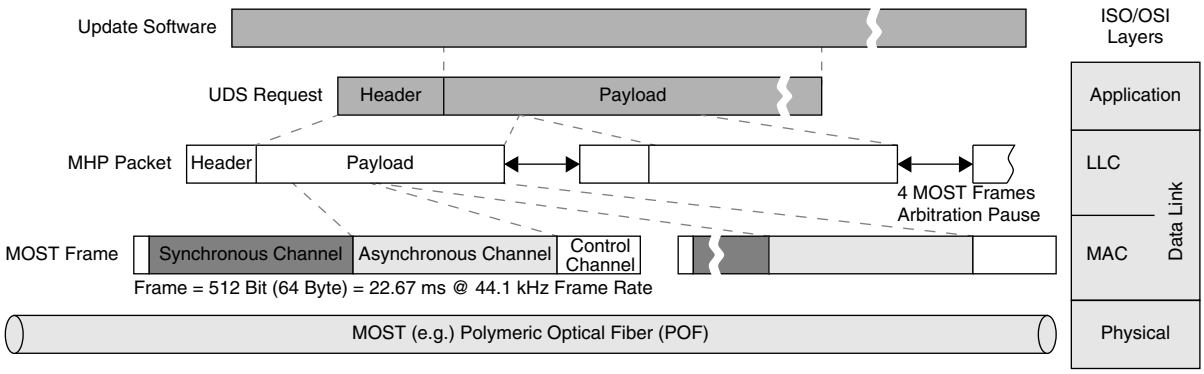

Fig. 1. Protocol stack and encapsulation of protocol data units for software updates of electronic devices.

into the flash memory, and informs the tester about the success of this operation by transmitting a *Transfer Data Response* message. The UDS protocol can use the transport services of MHP (see Figure 1). For this reason, a UDS message is segmented in several MHP packets and transferred as one MHP block.

D. Memory Types of a Device

Basically, an in-vehicle electronic device has different types of memory: There is volatile *Random Access Memory* (RAM) and *Read Only Memory* (ROM). Software stored in the ROM can never be updated throughout the whole life time cycle of a car. In the *Electrically Erasable Programmable ROM* (EEPROM) altering data (e.g. error messages) is stored. Software stored in the *flash memory* is persistent, too. In case of EEPROM and flash memory a new memory programming throughout the life time cycle of a car is possible.

E. Update Process

The software update of a single device can be divided in three major phases: (1) *Pre-Update phase*, (2) *Actual-Update phase*, and (3) *Post-Update phase*. During the Pre-Update phase, a new diagnostic session is initiated by a diagnostic tester. Afterwards, the device's hardware version and the actual installed software version are detected. Furthermore, the target device switches from normal operation mode into the memory-programming mode and the diagnostic tester has to authenticate itself to the target device.

During the Actual-Update phase, the target device receives consecutive software segments and writes them into its flash memory. A segment is transferred to the device by at least one Transfer Data Request. Therefore, there is usually the need to segment the update software. At first, a device receives packets of a segment and stores them in the RAM. After having completely received the segment, it is written from RAM into the flash memory in one step. Thus, the segment size is limited by the RAM size of the device.

During the Post-Update phase, the device validates the updated software and its history. Finally, the device switches back to the normal operation mode.

The total update time is significantly affected by the duration of the Actual-Update phase. The Pre-Update phase and Post-Update phase are negligible. Therefore, we analyze only the period of the Actual-Update phase activities.

III. Modeling Software Updates

We use colored Petri nets in order to model the concurrency and synchronization of the software update activities. As a result of the modeling, an evaluation of the efficiency of several algorithms that schedule these activities is possible.

A. Model on the Application Layer

As shown in Figure 2 the model is divided into three parts: the *Dispatcher*, the *Bus*, and the *Device*. A dispatcher, modeled by the transition $t_{\mathrm{Dispatcher}}$, can be implemented in a diagnostic tester. It transmits segments to the target devices.

The transition $t_{\mathrm{Dispatcher}}$ fires when every input place contains at least one token of the same color or an uncolored token. Each color corresponds to an appropriate target device. The firing models the actual transmission of a segment in the payload of a Transfer Data Request. The segments that are still to be sent are represented by the colored tokens located on the place *Pool*. The number of segments (number of tokens) for device i (color i) is $\theta_{\mathrm{Seg,i}} = \lceil s_{\mathrm{SW,i}}/s_{\mathrm{Seg,i}} \rceil$, where $s_{\mathrm{SW,i}}$ is the software size and $s_{\mathrm{Seg,i}}$ the segment size.[1] Every firing leads to a removal of the appropriate colored token on the place *Pool*.

The place *Schedule* represents the schedule. The tokens in this place stand for the segments that are allowed to be sent next according to the scheduling algorithm. After every firing, the next set of tokens for the place *Schedule* is computed. The tokens in the place *ReadyToReceive* symbolize by their color the devices that are ready to receive. As described in the previous section, a device is only ready for receiving a segment, if it is not busy because it is writing another segment into its flash memory.

As described in section II-C a diagnostic response actually informs the dispatcher about the readiness for receiving the

[1]We do not consider cuttings due to the segmentation of the software for device i into segments. For our analysis cuttings are negligible.

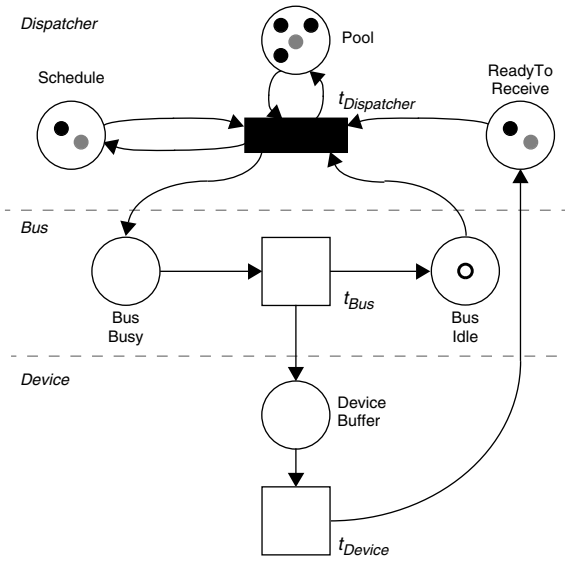

Fig. 2. Petri net of the update activities on the Application layer.

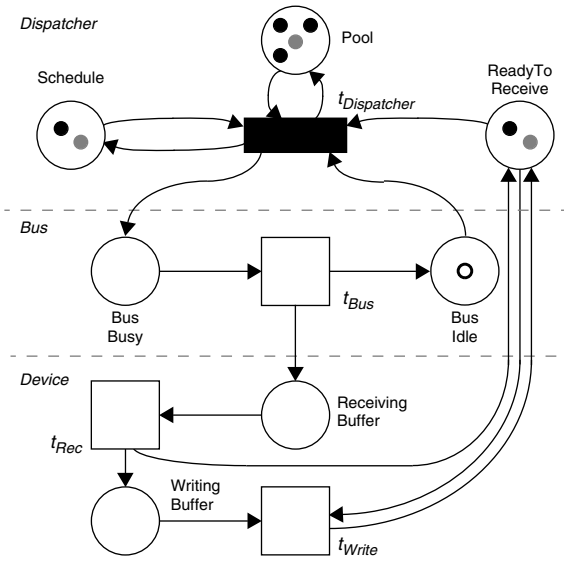

Fig. 3. Refined Petri net of the update activities on the LLC layer.

next diagnostic request. In our model, this behavior is represented by the migration of a token from place *DeviceBuffer* to the place *ReadyToReceive*. Therefore, the model does not provide additional bus occupancy by a diagnostic response. This is right because actually the response can be transmitted via the control channel. In this case, no resources of the asynchronous channel are occupied. The time impact of a response can be considered by the transition t_{Device}.

Finally, a segment can only be transmitted if the bus is idle. For this purpose, the place *BusIdle* contains an uncolored token.

The bus is modeled by the two places *BusBusy* and *BusIdle* and the timed transition t_{Bus}. The firing of this transition models the bus occupancy during a transmission of a segment. The delay of the transition depends on the data rate of the MOST bus, the device's receiving time of a segment, the segment size, and the sizes of the MHP packets.

By firing t_{Bus} an uncolored token is generated on place *BusIdle*. Furthermore, the colored token on the place *BusBusy* that stands for a segment migrates to the place *DeviceBuffer*. There, this token represents the existence of a segment in the RAM of a device that corresponds to its color. If there is a token in the input place *DeviceBuffer* the timed transition t_{Device} fires. This means that the segment is written into the flash memory.[2] After firing, the token migrates to the place *ReadyToReceive*.

As mentioned before a Transfer Data Request is transferred as one MHP block. Note that the model already considers the acknowledgment of the MHP block by a properly later firing of the transition t_{Bus}.

[2]In this paper writing a segment into the flash memory includes a previous deletion of the corresponding space of the flash memory.

B. Model on the LLC Layer

To take into account algorithms for scheduling MHP packets on the LLC layer, too, we have extended the previous model as shown in Figure 3. In this model, the tokens in the place *Pool* now represent MHP packets instead of software segments. The number of MHP packets (number of tokens) for device i (color i) is $\theta_{\mathrm{Packet,i}} = \lceil s_{\mathrm{Seg,i}}/s_{\mathrm{Packet}} \rceil * \ \theta_{\mathrm{Seg,i}}$, where s_{Packet} has a maximum length of 1006 bytes according to the maximum payload size of a MHP packet (see section II-B).[3]

Furthermore, the tokens in the place *BusBusy* represents MHP packets as well as the tokens in the place *Pool*. The acknowledgment of a MHP block is considered by means of a properly later firing of the transition t_{Bus} in case of the token that represents the last packet of a Diagnostic Request.

The main extension concerns the device part. In a device, MHP packets are received from the bus and at first are stored in the receiving buffer located on the NIC. This is modeled by the additional place *ReceivingBuffer*.

The receiving buffer on a NIC can contain only one MHP packet per device. Therefore, the place *ReceivingBuffer* contains a maximum of one token per color. The receiving delay $\tau_{\mathrm{Rec,i}}$ within a device is modeled by the timed transition t_{Rec} (see section I). Often, this time is caused by the EHC's polling with a constant rate. By firing this transition a colored token migrates from place *ReceivingBuffer* to place *WritingBuffer*. This models the storing of the packet payload in the RAM.

As soon as $\lceil s_{\mathrm{Seg}}/s_{\mathrm{Packet}} \rceil$ packets (tokens with the same color) are stored on the place *ReceivingBuffer*, the timed transition t_{Write} fires. This represents writing the reassembled segment into the flash memory. The writing time $\tau_{\mathrm{Write,i}}$ depends on the segment size $s_{\mathrm{Seg,i}}$ and the writing rate of the flash memory of device i.

[3]We do not consider cuttings due to the segmentation in packets. For our analysis cuttings are negligible.

After sending a MHP packet to a device it is ready to receive again. This is modeled by an additional arc from transition t_{Rec} to place *ReadyToReceive*.

While writing a segment into flash memory, a device is not capable to receive packets. For this purpose, an additional arc links from place *ReadyToReceive* to transition t_{Write}.

IV. Scheduling Algorithms

A *Scheduling Algorithm* for updating software of in-vehicle electronic devices defines the schedule for the transmission of diagnostic requests on the Application layer or MHP packets on the LLC layer, respectively.

This section introduces several scheduling algorithms that are then evaluated in section V. Due to the well-known parameters, for each algorithm the complete order of transmitting segments can be calculated before beginning the update process.

A. Consecutive

The state of the art is to update the devices in a consecutive manner and not simultaneously. In the former case each device is served completely one right after another. After all segments are sent to a device and written into its flash memory, the next one will be updated. Normally, the bus utilization is low due to the slowness of packet reception and the slowness of writing segments into the flash memory.

B. Scheduling on the Application Layer

On the Application layer, a scheduler knows the transmission order of diagnostic requests containing the software segments.

1) Immediate Send: The dispatcher that regards an *Immediate Send* scheduling is ready to send the next diagnostic request immediately when a corresponding device is not busy by flash memory writing, but ready to receive. This means that the dispatcher actually has to wait for a diagnostic response before it becomes ready to send the next diagnostic request.

It is possible that there is a duration between becoming the readiness to send and the actual sending due to the bus occupancy by the transmission of segments for other devices.

An Immediate Send scheduler is simply modeled in the Petri nets by the fact that the place *Schedule* contains always a token of each color. Consequently, the firing of transition $t_{\mathrm{Dispatcher}}$ depends only on the states of the places *Pool* and *ReadyToReceive*.

2) Round Robin: The dispatcher transmits diagnostic requests to each device in a periodically repeated order. After a device is updated completely, it is deposited from the *Round Robin* (RR) process. It is possible that the dispatcher is blocked if there is a diagnostic request to send to an already busy device. The blocking is broken as soon as the dispatcher is informed about the readiness for receiving of this device by a diagnostic response. A RR scheduler is modeled by the fact that the place *Schedule* contains only one token at all times. By each firing the color of the token is alternating.

3) Weighted Round Robin: Generally, there are differences regarding to the single update times of the devices. Consequently, there is always an update of a single device that remains alone at the end. In order to decrease the total update time this device should be served more often at the beginning. This is possible by introducing a *Weighted Round Robin* (WRR) algorithm that prefers appropriate devices.

The following approach considers the number of segments $\theta_{\mathrm{Seg,i}}$ for each device i. The idea is to find an order pattern that distributes the transmission of segments homogeneously.

Therefore, we introduce a number n_i per device i with $n_i = 1$ as initial value for all i. We get the ratios $r_i = \theta_{\mathrm{Seg,i}}/n_i$. The dispatcher has to serve the device j with maximum r_j. Afterwards we increment n_j and repeat the calculation of the ratios. Due to this repetition further devices have to be served.

For instance, there are three devices with $\theta_{\mathrm{Seg},1} = 153$, $\theta_{\mathrm{Seg},2} = 100$, and $\theta_{\mathrm{Seg},3} = 67$. In the first step the ratios are $r_1 = 153/1, r_2 = 100/1, r_3 = 67/1$. We choose device 1. In the second step the ratios are $r_1 = 153/2, r_2 = 100/1, r_3 = 67/1$. The maximum value is r_2 and therefore we choose device 2. In the next step the ratios are $r_1 = 153/2, r_2 = 100/2, r_3 = 67/1$. Now, the maximum value is $r_1 = 153/2$ and therefore we choose device 1 again. This algorithm continues until all segments are transmitted. After all segments of a device k are scheduled n_k is set to infinite.

In addition, there is a special rule: The algorithm tries to swap contiguous elements in order to avoid direct consecutive transmissions for the same device.

4) Greedy: The idea of a *Greedy* algorithm is to make always the choice that looks best at the moment [6]. In our case, in each step the device i with the largest remainder update time $\max\{\tau_i(u)\}$ at time u will be served as next. This remainder update time is calculated as

$$\tau_i(u) = \theta_{\mathrm{Seg,i}}(u)\left(\tau_{\mathrm{Write,i}} + \frac{s_{\mathrm{Seg,i}}}{s_{\mathrm{Packet,i}}}\left(\tau_{\mathrm{Rec,i}} + \tau_{\mathrm{Trans,i}}\right)\right) \tag{1}$$

with

$$\tau_{\mathrm{Trans,i}} = \frac{\left\lceil \frac{s_{\mathrm{Packet,i}}+18}{36} \right\rceil + 4}{44.1\ \mathrm{kHz}}, \tag{2}$$

where $\tau_{\mathrm{Trans,i}}$ is the transmission time for one MHP packet. The applied MOST- and MHP-specific values are introduced in section II. The parameter $\theta_{\mathrm{Seg,i}}(u)$ describes the remainder number of segments at time u.

During the calculation, the rule to swap contiguous elements in the determined order as above mentioned has to be applied, too.

C. Scheduling on the LLC Layer

The diagnostic application segments the software and moves the segments down to the LLC layer. There, each segment is encapsulated in a MHP block. The MHP is responsible for the reliable transmission of every MHP packet of the block to the target device. Interleaving of MHP blocks enables a simultaneous transmission to multiple devices. This feature

TABLE I

OVERVIEW OF THE PARAMETER SETS

Device	$s_{\mathrm{SW,i}}$ [Mbytes]	$s_{\mathrm{Seg,i}}$ [kbytes]				$\tau_{\mathrm{Rec,i}}$ [ms]				$\tau_{\mathrm{Write,i}}$ [ms]			
		S.1	S.2	S.3	S.4	S.1	S.2	S.3	S.4	S.1	S.2	S.3	S.4
TV receiver	20	16	16	16	16	15	15	10	15	160.96	160.96	160.96	321.92
Navigation system (without map data)	20	16	64	16	16	2	2	10	2	160.96	643.84	160.96	160.96
Mobile phone interface	1	16	1	16	16	10	10	10	10	321.92	20.12	321.92	321.92
Audio amplifier	1	16	4	16	16	10	10	10	10	321.92	80.48	321.92	1287.68
iPod interface	1	16	8	16	16	10	10	10	10	1287.68	643.84	1287.68	6438.4

is actually realized by means of a MOST network driver enhancement. Thereby, a RR strategy is applied to send MHP packets to multiple devices simultaneously. In this context, the RR is non-blocking. The available MHP blocks are examined one by one and in each case a MHP packet is transmitted.

Certainly, scheduling on the LLC layer is only applicable if diagnostic requests (software segments) on the Application layer are multiplexed. Therefore, a scheduling algorithm on the Application layer has to be applied. In this paper we simply apply *Immediate Send*.

V. PERFORMANCE EVALUATION

The models described in section III were implemented by *CPNTools* [7]. This tool supports the analysis of timed, colored Petri nets. As mentioned before, the scheduling algorithms are implemented in the transition $t_{\mathrm{Dispatcher}}$ that calculates a new state of the place *Schedule* while firing.

We assume a multimedia and infotainment domain composed by five devices that have to be updated. The evaluation criterion is the obtained total update time. We define four scenarios (*S.1 – S.4*) by varying the device parameters as listed in Table I.

In all scenarios the both devices TV receiver and navigation system have by far the largest amount of update software. In *S.1*, the receiving performance of the navigation system is the best, and the receiving performance of the TV receiver is the worst. Furthermore, the iPod interface device has the worst flash memory writing time. In *S.2*, we vary only the segment sizes. The other parameters are not changed. This means that in particular the writing rates of the flash memories are unvaried. In *S.3*, we modify the receiving times to identical values. Finally, in *S.4* we change the flash memory writing times. In this scenario the flash memory of the iPod interface device performs badly.

The MHP packets carry always the maximum amount of 1006 bytes. Thereby, $\tau_{\mathrm{Trans,i}}$ is approximately 0.77 ms according to (2). However, we assume a transmission time of 1 ms due to an additional delay of the transmission driver.

A. Algorithms on the Application layer

At first, the process of evaluation confirms our expectation that any simultaneous update has a significant better performance than a consecutive update. Furthermore, we see that the more parameters an algorithm for a simultaneous update process takes into account, the more the total update time decreases. The main problem is to find a compromise between them in order to decrease the update time as much as possible.

TABLE II

TOTAL UPDATE TIME [MM:SS]

Algorithm (Application layer)	S.1	S.2	S.3	S.4
Consecutive	16:00	16:00	17:01	25:57
Immediate Send	10:00	11:09	9:20	18:40
RR	10:01	11:10	8:46	18:42
WRR	9:37	10:46	8:55	17:09
Greedy	9:15	10:32	8:46	15:40

TABLE III

TOTAL UPDATE TIME [MM:SS]

Algorithm (LLC layer)	S.1	S.2	S.3	S.4
Consecutive	16:00	16:00	17:01	25:57
RR	8:54	8:54	7:11	12:20

The Immediate Send algorithm considers no parameters, hence applying it does not result in the shortest update times in none of the scenarios. The RR algorithm leads mostly to undesired results because it only takes into account the order of the devices. Therefore, there is often an unfair distribution of the transmission of the segments. The WRR algorithm tries to achieve a fair distribution. In most cases, this trial is successful (see *S.1*, *S.2*, and *S.4*). However, *S.3* shows that WRR can perform worse than RR.

In all scenarios, the WRR leads to an alternation between serving the TV receiver and the navigation system. The longer receiving times of the navigation system in *S.3* (10 ms instead of 2 ms) leads to longer serving times. This means that the WRR algorithm blocks for a longer time even though all other devices could be served. In contrast, the advantage of the RR algorithm is that at least at the beginning all devices can be served without blocking.

The scenarios show that the implemented Greedy algorithm that considers the remaining update time, is the best choice in most of the cases, although blocking can also occur. Due to this blocking behavior, the Greedy algorithm and the RR algorithm even achieve in *S.3* the same total update time.

B. Algorithms on the LLC layer

The above mentioned scenarios are again used to evaluate the performance of the RR algorithm on the LLC layer. On the Application layer the Immediate Send algorithm is applied. Table III lists the results for both the Consecutive algorithm and the RR algorithm on the LLC layer.

In all scenarios, the RR algorithm achieves the best results in comparison to scheduling algorithms on the Application layer.

The reason is that the devices have large packet receiving times in comparison to the transmission times. Therefore, transmitting packets of a segment consecutively causes frequent bus idle times. The RR algorithm multiplexes packets for multiple devices and consequently reduces the bus idle times.

VI. Conclusion and Outlook

The amount of in-vehicle software stored in flash memories increases rapidly. Therefore, future software updates have to be more efficient. In this paper, we investigated and evaluated scheduling algorithms that can be applied in order to reach this aim.

We described technologies, protocols and processes that can be used for updating software. In passenger cars MOST is commonly used to interconnect devices in the multimedia and infotainment domain. MHP supports a reliable transfer of the update data to the target devices via MOST. A target device generally offers a Transfer Data Service that is part of UDS in order to enable its updatability.

We analyzed scheduling algorithms working both on the Application layer and on the LLC layer. The results show that the Greedy algorithm performs as best on the Application layer. It has to take into account many parameters in order to be efficient. However, in practice these parameters are often critical because their values can substantially vary, and therefore be insignificant. For instance, the flash memory writing rates vary due to impacts caused by, e.g., temperature, age, and rewrite frequency. Nevertheless, the total update time can be decreased by applying an intelligent scheduling algorithm if the deviation of critical values is limited.

The most promising approach is to apply a combination of a simple algorithm for multiplexing diagnostics' requests on the Application layer (like Immediate Send) and RR on the LLC layer. As shown in this paper, this combination leads commonly to short update times without concerning additional parameters.

We consider the software update of electronic devices interconnected by MOST. Some conclusions seems to be valid for other in-vehicle networks, too, e.g., *Controller Area Networks* (CAN) [8] and *FlexRay* [9] bus systems. For CAN and especially FlexRay the devices and not the bus system are often the bottleneck. Therefore, updating devices simultaneously can improve the performance significantly. In the next steps, we will investigate the software updates in further in-vehicle networks.

The introduced scheduling algorithms can be used for updating the full software that is stored in the flash memory of a device as well as only parts of them. Updating parts of the software does not have an impact on the efficiency of the considered scheduling algorithms. An incremental update process decreases only the software sizes. Methods to identify affected software parts are out of the focus of this paper. For instance, they are considered as methods for version control and configuration management in [10].

In addition to a simultaneous software update, further methods could be applied. For instance, the large amount of software can be reduced by using compression. In the future, we will focus on a hybrid approach that combines compression and simultaneous software updates.

Acknowledgments

The authors would like to thank Michael Scharf for his helpful criticisms and suggestions.

References

[1] M. Broy, I. Krüger, and M. Meisinger, *Automotive Software-Connected Services in Mobile Networks: First Automotive Software Workshop*. Springer, 2004.

[2] MOST Cooperation, *MOST Specification – Version 2.5-00*, October 2006.

[3] A. Grzemba, *MOST. Das Multimedia-Bussystem für den Einsatz im Automobil*. Franzis, 2007.

[4] MOST Cooperation, *MOST High Protocol Specification – Version 2.1-01*, February 2001.

[5] International Organization for Standardization, *ISO/DIS 14229-1 Road Vehicles – Unified Diagnostic Services – Part 1: Specification and Requirements*, October 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.

[7] CPN Group, University of Aarhus, "CPN Tools – Computer Tool for Coloured Petri Nets," wiki.daimi.au.dk/cpntools/, Denmark, 2007.

[8] International Organization for Standardization, *ISO 11898-1:2003 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, November 2003.

[9] FlexRay Consortium, *FlexRay communication system – protocol specification – version 2.1*, December 2005.

[10] C. Heinisch, V. Feil, and M. Simons, "Efficient configuration management of automotive software," in *2nd European Congress ERTS Embedded Real Time Software*, January 2004.