Message Reordering for the Reuse of CAN-based Legacy Applications in a Time-Triggered Architecture

R. Obermaisser
Vienna University of Technology, Austria
email: ro@vmars.tuwien.ac.at

***Abstract*— While CAN is the most widely used communication protocol in present day distributed automotive computer systems, time-triggered architecture will provide the ability to handle the communication needs of future by-wire cars. In addition to hard real-time performance, time-triggered architectures help in managing the complexity of fault-tolerance and corresponding formal dependability models, as required for the establishment of ultra-high reliability (failure rates in the order of 10^{-9} failures/hour). Virtual CAN networks on top of a time-triggered communication protocol are a solution to integrate existing CAN-based legacy applications into such a time-triggered architecture. Thus, there is the possibility to eliminate physical CAN networks, which leads to cost reductions and reliability improvements. In order to ensure that existing CAN-based software works correctly in a time-triggered architecture, a virtual CAN network must provide the temporal behavior of a physical CAN network. For this reason, we develop a solution for establishing in a virtual CAN network the same temporal message order as in a physical CAN network. We present a CAN protocol emulation algorithm and provide validation results based on an implementation in the Time-Triggered Architecture.**

I. Introduction

Today, the event-triggered CAN protocol [1] is by far the most widely used automotive protocol with a 100% market penetration. Major advantages of CAN include its high flexibility (e.g., strong migration transparency, no need to change a communication schedule when adding Electronic Control Units (ECUs)), resource efficiency through the sharing of bandwidth between ECUs, and high availability of CAN-based tools and engineers with CAN know-how. Despite the use of Time-Triggered (TT) architectures in future by-wire cars, CAN is likely to remain as a communication protocol for non-safety critical subsystems due to the higher flexibility and average performance. Even for safety-related subsystems, CAN-based legacy applications will not be replaced instantly.

Event Channels (ECs) for the dissemination of CAN messages in a time-triggered architecture are a solution for layering Event-Triggered (ET) CAN communication on top of a TT communication protocol [2]. An EC is an architectural service that accepts as its input transmission requests from an ET application at arbitrary instants, thus improving the flexibility of a time-triggered architecture. In addition to periodic time-triggered message exchanges, this service permits to communicate without having to statically predefine the instants of all message exchanges. Liberating non safety-critical application subsystems from the need to specify a static communication schedule at design time simplifies future modifications of the application software. If additional or different messages are sent via an EC, the underlying time-triggered communication schedule can remain unchanged. Furthermore, multiple messages sent by a node can share bandwidth on the network in order to improve resource efficiency.

ECs enable the integration of applications for which a CAN communication service is preferable, as well as the integration of CAN-based legacy applications into a TT computing platform. ECs for CAN are an effective solution in many problem domains, such as diagnosis and maintenance functionality or newly developed non safety-critical applications. However, for reuse of CAN-based legacy applications with ECs, it is important to note that ECs can exhibit a different temporal order of received messages than a physical CAN network for the same sequence of message transmission requests. While this difference may not be a concern in many newly developed applications, it poses a problem for the reuse of CAN-based legacy software. Substantial re-testing or adaptations of existing code would be required to ensure a correct behavior of legacy software despite the different temporal message order compared to the platform the application software has been developed for. The need for adaptation is a particular problem, if source files are unavailable due to Intellectual Property (IP) protection reasons and changes must be back-propagated to the suppliers. Clearly, in such a situation modifications of the communication system in order to ensure correct behavior of ECUs is preferable to the opposite solution of ECU modifications to migrate them to a new communication system.

For this reason, we have developed a solution for establishing in ECs the temporal message order of a physical CAN network. The result is a *Virtual CAN Network (VCN)* as an overlay network on top of a TT communication protocol for the reuse of legacy applications without the need for redevelopment. A VCN preserves past investments by leveraging results from previous ECU tests, integration tests, and field data (e.g., failure free operation in a large number of deployed cars). We take ECs as introduced in [3] as a starting point and execute a distributed protocol emulation algorithm for reordering messages. The proto-

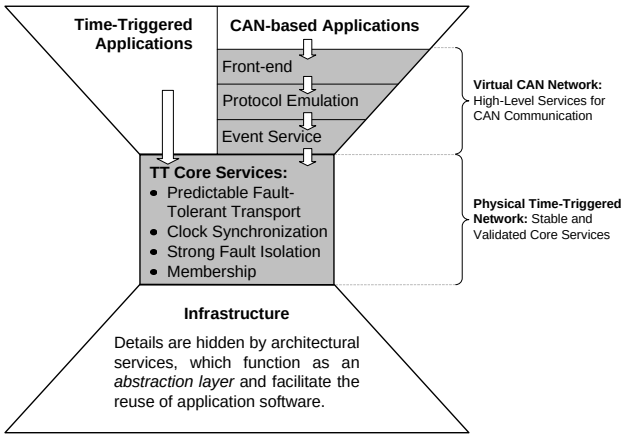

Fig. 1. Architecture with High-Level CAN Communication Services

col emulation algorithm exploits the global time base of the underlying TT architecture in order to assign to each message a timestamp denoting the transmission request instant. In each node, these timestamps in combination with the message identifiers and message lengths form the input for a dynamic simulation of the Carrier Sense Multiple Access Collision Avoidance (CSMA/CA) media access control strategy.

This paper is structured as follows. The construction of ECs as part of an integrated architecture for ET and TT control is the focus of Section II. In Section III, we discuss differences in the temporal message order in ECs and physical CAN networks. Examples demonstrate that the same transmission request instants can cause a different order of receive instants. Section IV presents a protocol emulation algorithm for reordering messages, thus establishing a VCN. Thereby, the temporal order of the message receive instants is made identical to the order in a physical CAN network. We discuss the results of validation activities based on an implementation of a VCN with protocol emulation in the Time-Triggered Architecture (TTA) in Section V. With protocol emulation we have observed no difference with respect to the message ordering between the virtual and physical CAN networks.

II. Integrated Architecture for Time-Triggered and CAN-based Applications

By providing CAN communication services in the TTA, we establish an integrated architecture for ET and TT control [3]. This integrated architecture is designed for mixed-criticality systems up to the highest criticality class (e.g., SIL4 in EN ISO/IEC 61508 [4]). The integrated architecture supports two subsystems, namely a TT subsystem and a CAN subsystem. Safety-critical functionality is always realized in the TT subsystem, while the CAN subsystem offers to non safety-critical applications an execution environment with high flexibility and average performance. The underlying design decision of using the TT control paradigm for safety-critical functionality conforms to widely accepted requirements for the infrastructure of safety-critical real-time systems [5], [6].

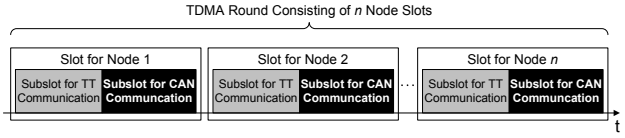

Fig. 2. Temporal Subdivision of TDMA Slots

As depicted in Figure 1 the integrated architecture is a waist-line architecture, which distinguishes between *core services* and *high-level services* that build on top of the core services. The core services are a common basis for the TT subsystem and the high-level services of the CAN subsystem. The core services include a predictable TT message transport service, fault tolerant clock synchronization, strong fault isolation, and consistent diagnosis of failing nodes through a membership service.

The high-level services realize the communication infrastructure for the CAN subsystem as a *Virtual CAN Network (VCN)*, which is an overlay network on top of the TT physical network. For the realization of the VCN, we perform a temporal subdivision of the communication resources provided by the Time Division Multiple Access (TDMA) scheme of the TT physical network. TDMA statically divides the channel capacity into a number of slots and assigns to each node a unique slot that periodically reoccurs at a priori specified global points in time. We subdivide each node's slot into two subslots, namely a slot for TT communication and a slot for the ET dissemination of CAN messages (see Figure 2).

A. Event Service

The *event service* is a high-level service that exploits the subslot for CAN communication in order to establish an *Event Channel (EC)* for the on-demand transmission of CAN messages. The event service maps an ET communication service to the sparse time base [7] of a TT communication service. Although message transmission requests can occur at arbitrary instants, the dissemination of the messages on the underlying TT network always is performed at the predefined global points in time of the CAN subslots. Outgoing messages are buffered in message queues until the respective node's subslot for CAN communication occurs in the TDMA scheme. Also, the queuing of messages handles bursts during which the bandwidth consumption of outgoing messages exceeds the bandwidth that is available via CAN subslots. In every node, the event service performs a fragmentation of outgoing messages into packets that can be placed in the node's CAN subslot. In addition, the event service reassembles messages out of received packets.

B. Protocol Emulation

The high-level service for protocol emulation exploits the ECs provided by the event service in order to realize a Virtual CAN Network (VCN). The protocol emulation establishes the temporal message order of a physical CAN network by performing at run-time a simulation of the CSMA/CA media access control strategy of a physical

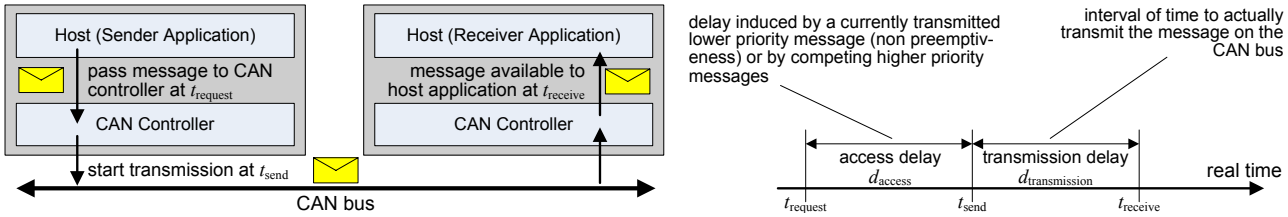

Fig. 3. Message Delay induced by a CAN Communication System

CAN network within each node of the TT system (see Section IV).

C. Front-end

The third high-level service establishes the *Application Programming Interface (API)* by which CAN-based application software can access a VCN. For this purpose, this high-level service implements the register interface of commercial CAN controllers (e.g., i82527 [8]). Thereby, the API permits to reuse CAN-based legacy application software that has been designed to directly access the respective CAN controller at the register level. In addition, the functionality of this high-level service comprises message filtering in order to selectively receive messages based on identifier masks.

III. Temporal Order of CAN Messages

In a distributed system the sequence of message transmissions on the communication system results in a context for each message, because earlier messages can influence how a message needs to be interpreted. For this reason, different types of message orders (e.g., atomic broadcast [9]) have been defined. Each message order provides respective properties that reflect the underlying assumptions concerning what temporal and consistency properties are relevant for correctly interpreting messages at the receivers. In this paper, we focus on the temporal order of the message receive instants, called *temporal message order* for short. The temporal order is the order of the instants on the timeline [10].

A. Physical CAN Network

If the application software in a physical CAN node requests the transmission of a message m at instant t_{request} (see Figure 3), the dissemination of m will be

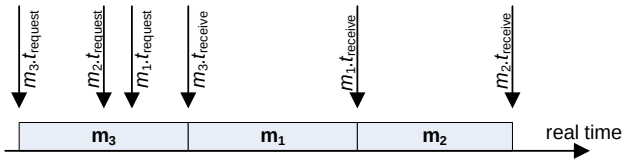

Fig. 4. Example for Message Order in a Physical CAN Network

delayed until the bus becomes idle and m wins in the arbitration process. Let t_{send} denote the instant when the successful message transmission is started, i.e. without a later abortion due to lost arbitration. We call the delay between the request instant t_{request} and the send instant t_{send} the access delay d_{access}. If the bus is idle at t_{request} and the message exhibits the highest priority among all pending messages, then the message will be sent immediately. In this case t_{send} equals t_{request}. Otherwise, the message remains pending until the bus becomes idle and the message wins in the arbitration process.

We denote the instant when the complete message (i.e. the last bit of the CAN frame) has arrived at the receiver the receive instant t_{receive} of the message. The transmission delay $d_{\text{transmission}}$ is the delay between the send instant t_{send} and the receive instant t_{receive}.

Based on the request instants t_{request} as inputs to the communication system, we can look at the resulting temporal order of the receive instants t_{receive}. Figure 4 contains an example for the temporal order of the receive instants based on the request instants of three messages m_1, m_2, and m_3. The subscripts denote the priority, i.e. m_1 is the message with the highest priority. Since CAN is non-preemptive, the started transmission of message m_3 with the lowest priority is completed despite the transmission requests for the higher priority messages. When the bus becomes idle, the CAN arbitration mechanism results in the transmission of m_1 prior to m_2.

B. Event Channel

If an event service as described in Section II is used for the dissemination of CAN messages, Figure 5 shows a

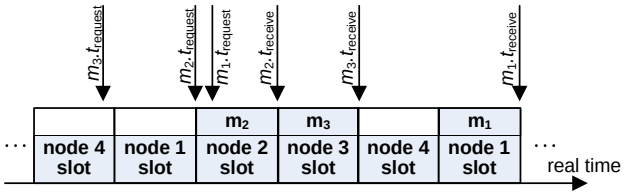

Fig. 5. Example for Message Order at an Event Channel

possible temporal order of receive instants. This scenario uses the same sequence of transmission requests as in the previous example for the physical CAN network and assumes an exemplary phase-shift relationship between the message transmission requests and the underlying TDMA scheme. The distributed system consists of four nodes in this example and employs a TT communication schedule that allows each node to send exactly once within a TDMA round. Also, to keep the example simple we assume that each message m_i ($i \in \{1, 2, 3\}$) is sent by node i and CAN slots are sufficiently large to avoid the need for fragmentation. Note, however, that

the differences w.r.t. the message order can also occur without these restrictions. The example scenario results in a different message order compared to the physical network. Message m_2 exhibits the smallest access delay, because the slot of node 2 immediately succeeds the transmission request. The lowest priority message m_3 is transmitted next with an access delay of two slots. Message m_1 experiences an access delay of three slots.

The reason for the different temporal order of the receive instants compared to a physical CAN network are the access delays caused by the underlying TT communication system. The differences originate from two properties:

- **Sampling of messages.** An EC maps a dense time base to a sparse time base, because the TT core transport service requires all message transmissions to occur in statically assigned time slots. Since the ET CAN communication activities are not synchronized to this TDMA scheme, the access delays are variable and can include the duration of a complete TDMA round until the slot of the respective node reoccurs. The access delay of a message depends on the instant of the transmission request relative to the start instant of the node's slot in the TT communication schedule. For example, in experiments for measuring the transmission latencies of ECs, the used communication schedule has resulted in worst-case sampling delays of 320 μs [11].
- **No contention between nodes.** Messages from different nodes do not dynamically compete for the shared medium. All conflicts are resolved with a static schedule. Consequently, the message priorities (specified via the message identifiers) of two messages sent at separate nodes have no influence on the message send instants. The two nodes do not share bandwidth and thus there is no dynamic decision on which message to transmit first. The send instant of each message is determined locally by the other transmission requests at this node. While this non-interference between nodes is beneficial from a fault isolation and composability perspective (i.e. a constructive integration of a distributed system), the missing contention between nodes also implies that legacy applications can perceive a different message order compared to a physical CAN network.

IV. CAN Protocol Emulation

The CAN protocol emulation is a high-level architectural service that aims at the reuse of legacy applications with a minimum of redevelopment and retesting efforts. The CAN protocol emulation ensures that a VCN exhibits the same temporal order of the receive instants as a physical CAN network provided that the virtual network gets as input the same set of messages, in particular with identical transmission request instants. For this purpose, we employ a *protocol emulation algorithm* that is executed in every node to perform a simulation of a physical CAN network. This algorithm uses as inputs both messages for which a

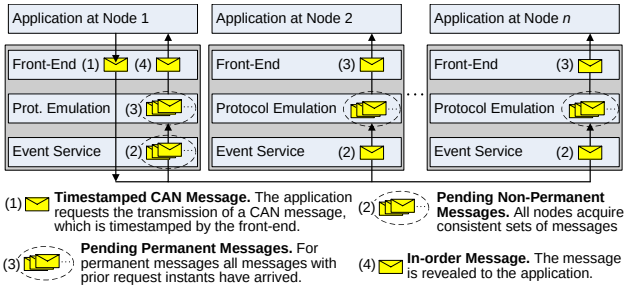

Fig. 6. Phases of a CAN Message Transmission

transmission has been requested by the local application (i.e. at the same node) and messages received via the event service from other nodes. The protocol emulation algorithm takes into account the request instant, the message priority, and the message length. However, the correctness of a legacy CAN application will most likely not depend on the temporal ordering induced by communication faults and message retransmissions. Hence, we abstract from fault-effects on the temporal order of the message receive instants, because we are interested in providing an authentic communication system for the integration of CAN-based legacy applications.

Based on these inputs, the protocol emulation algorithm computes for each message the send instant when the message would have been sent on a physical CAN network. Messages are passed to the application in the order of ascending message send instants. Due to the non-preemptive nature of CAN, this strategy also ensures ascending message receive instants and thus the correct temporal message order.

The protocol emulation algorithm is fully decentralized and runs in all nodes participating in the CAN emulation. It forms the high-level architectural service for protocol emulation, which exchanges CAN messages with its adjacent service layers, namely the front-end and the event service.

A. Phases of a CAN Message Transmission

The transmission of a CAN message is requested at the front-end, which provides the API to the application software. Based on the information provided by the application (e.g., identifier, data bytes), the front-end builds a timestamped CAN message (step 1 in Figure 6) that has a syntax as depicted in Figure 7. The length field

Entry	Size	Meaning
length	8 bit	message size
$t_{request}$	32 bit	timestamp of request instant
request	1 bit	request or normal message
identifier-type	1 bit	length of identifier (11 or 29 bit)
cancellation	1 bit	request for message cancellation
identifier	11 or 29 bit	message name and priority
data	0-8 bytes	0-8 bytes of data

Fig. 7. Timestamped CAN Message.

specifies the size of the message, which is between 7 and 17 bytes. The timestamp t_{request} stores the transmission

request instant w.r.t. the global time base established by the core service for clock synchronization, i.e. the instant at which the front-end at the sending node was handed over the message for being sent. The request flag serves the purpose of triggering the transmission of the message with the specified identifier at the receiver. The identifier type flag is used to distinguish between 11-bit and 29-bit identifiers. The cancellation request flag informs the protocol emulation at receiving nodes about the cancellation of the message with the specified identifier. This flag enables a sender application to abort the delivery of a message to the receiver applications. A cancellation is only possible while the message is pending at the protocol emulation (see below), i.e. before the transmission would have started on the physical CAN network at instant t_{send}. The last elements of a timestamped CAN message are the identifier, which denotes both the message name and the message priority, and the data field.

In the second step (see Figure 6), the timestamped CAN message is broadcast to all other nodes of the distributed system via the event service. The timestamped CAN message informs the protocol emulation at other nodes about the message transmission request. In addition, the front-end forwards the timestamped CAN message to the local protocol emulation, i.e. the protocol emulation located at the node at which the transmission request has been issued. In conjunction with the consistency properties of the TT core communication service (which is the foundation for the event services and the exchange of CAN messages), the protocol emulation at each node acquires a consistent set of messages.

The timestamped CAN messages arriving at a node's protocol emulation service are not immediately revealed to the front-end and the application, but remain pending until the correct message order can be established. Pending messages are stored in intermediate data structures of the protocol emulation. In case a cancellation for a pending message (denoted by the cancellation flag) arrives, the message is discarded.

Before a received message is forwarded to the front-end, the message goes through the following three steps:

- **Pending Non-Permanent.** A message m with a transmission request instant t_{request} is non-permanent, if future messages, i.e. messages that have not yet arrived at the protocol emulation, can exhibit an earlier request instant than m. The notion of message permanence is based on the definition in [12]. Since messages with an earlier request instant can precede m in the temporal message order, message m must become permanent before it can be used in the simulation.
- **Pending Permanent.** After passing the *permanence test* described in Section IV-B, a pending message becomes permanent. For a *permanent message*, it is ensured that all future messages will possess later request instants. The simulation determines the temporal message order based on the pending permanent message as its input.
- **In-Order.** After a pending message has been sent in the simulation of the physical CAN bus (see Section IV-C), the message is denoted as *in-order*. The message is forwarded to the front-end to reveal it to the application software.

B. Permanence Test

A message m_1 is *permanent* at instant t_p, if it is known that no message m_2 with an earlier or equal request instant ($m_2.t_{\text{request}} \leq m_1.t_{\text{request}}$) can be received at a later instant t ($t > t_p$) via an EC.

For determining permanence, we exploit the fact that transmission request timestamps of messages received via an EC are monotonically increasing. The monotony of the transmission request timestamps results from the fact, that for every EC there is only a single sender, which exclusively sends messages via the EC.

In order to determine the permanence of messages, the protocol emulation maintains a vector $\overrightarrow{t_{\text{latest}}}$, which contains a 32-bit integer for every sender. The ith element of this vector contains the message request instant of the most recent message received from sender i. Since the request instants of messages from a particular sender are monotonically increasing, the element associated with the sender in $\overrightarrow{t_{\text{latest}}}$ represents a temporal bound for subsequent messages, i.e. all future timestamped CAN messages must contain a later request instant.

The comparison of request instants must take into account wraparounds of the 32-bit global time. We can establish a relation $\lhd$ for the comparison of timestamps based on the assumption that the difference of the "ideal" (infinite range) timestamps is always below 2^{31}:

$$t_1 \lhd t_2 \leftrightarrow \big(0 < t_2 - t_1 < 2^{31}\big) \vee \big(0 < t_2 - t_1 + 2^{32} < 2^{31}\big)$$
$$t_1, t_2 \in \big\{0, 1, \ldots, 2^{32} - 1\big\}$$

Furthermore, we employ in the permanence test the global consistent membership vector $\overrightarrow{g}$ provided by the TT core architecture in order to exclude faulty nodes.

$$\overrightarrow{g} = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{pmatrix} \in \{0,1\}^n \text{ with } g_i = \begin{cases} 1 & \text{node } i \text{ correct} \\ 0 & \text{otherwise} \end{cases}$$

A sufficient condition for the permanence of a message is that its request instant is earlier (with respect to $\lhd$) than all temporal bounds for message request instants of correct nodes in the vector $\overrightarrow{t_{\text{latest}}}$:

$$\bigwedge_{i=1}^{n} \big(m.t_{\text{request}} \lhd t_{\text{latest},i} \vee g_i = 0\big) \rightarrow m \text{ permanent} \quad (1)$$

where n is the number of nodes in the system.

C. Message Ordering

The temporal ordering of messages occurs through a simulation of a physical CAN network, where simulated message transmissions represent the simulation steps. The current simulation time is specified by the instant $t_{\text{idlestart}}$. $t_{\text{idlestart}}$ is a special instant that separates the messages which have been sent on the simulated CAN

```
1   // Variable description
2   //   t_latest[]          :  vector containing most recent request instants
3   //   g[]                 :  membership vector
4   //   H_nonperm           :  heap of non-permanent pending messages
5   //   H_perm              :  heap of permanent pending messages
6   //   t_idlestart         :  start of idle interval on simulated CAN bus
7   //   t_request           :  transmission request instant of a message

8   when (front-end → m = ⟨t_request, id, data, this node, 0⟩)
9     t_latest[this node] = m.t_request
10    insert m into H_nonperm
11    m → event service

12  when (event service → m = ⟨t_request, id, data, sender node, 0⟩)
13    t_latest[sender node] = m.t_request
14    insert m into H_nonperm

15  // Permanence test
16  when ( H_nonperm≠ ∅ ∧ ∃n for which t_latest[n] or g[n] has changed)
17    // get non permanent message with smallest timestamp
18    get message m = ⟨t_request, priority, data, 0⟩ from top of H_nonperm
19    if ( ∀n (m.t_request ◁ t_latest[n] ∨ g[n] = 0) )
20      // m is permanent
21      remove m from H_nonperm
22      insert m into H_perm

23  // Message Ordering
24  when ( H_perm≠ ∅ ∧ ∄n t_latest[n] ◁ t_idlestart)
25    get message m = ⟨t_request, priority, data, d_access⟩ from top of H_perm
26    d_transmission = transmission duration of m
27    remove m from H_perm
28    m → front-end
29    t_idlestart = max(m.t_request, t_idlestart) + d_transmission
30    for every message m_i = ⟨t_request, priority, data, d_access⟩ ∈ H_perm  do
31      if (t_idlestart ◁ m_i.t_request) m_i.d_access = 0
32      else m_i.d_access = t_idlestart − m_i.t_request
33    reorder H_perm
```

Fig. 8. Overview of Protocol Emulation Algorithm

bus from those that have not. $t_{\text{idlestart}}$ marks the beginning of idleness on the simulated CAN bus. The message transmissions before $t_{\text{idlestart}}$ are already fixed, i.e. no later transmission requests can result in a modification of the sequence of message transmissions. Consequently, $t_{\text{idlestart}}$ also separates the ordered messages from the non-ordered ones.

In case the simulation time lies before the minimum request instant of a future timestamped CAN message ($t_{\text{idlestart}} \lhd \min_i(t_{\text{latest},i})$) and one or more pending permanent messages are available, a simulation step can be taken. Out of the set of pending permanent messages, the protocol emulation chooses a message for the next simulation step based on the request instants and the message priorities. After the simulation step, the selected message becomes in-order and is transferred from the protocol emulation to the CAN front-end. Simulation steps are executed until no more pending permanent messages are available or a future timestamped CAN message can exhibit an earlier request instant than the current simulation time.

D. Algorithm

The protocol emulation service performs the permanence test and the reordering of messages via the algorithm in Figure 8. This algorithm operates on two data structures, namely a heap of non-permanent pending messages and a heap of permanent pending messages. The elements of this heap are timestamped CAN messages. The primary key used for ordering in this heap is the sum of the message's request instant and the message's access delay (with respect to $\lhd$). The secondary key is the message priority (identifier). The reason for selecting heaps as the data structures for pending messages is the need for the repeated retrieval of messages with the smallest request instant plus access delay and highest priority.

A message is inserted into the heap of non-permanent pending messages either when a message arrives from the network (i.e. the event service passes the messages to the protocol emulation in line 12) or after the application software has issued a transmission request at the front-end (line 8). In both cases, the front-end at the sending node has already set the transmission request timestamp of the timestamped CAN message.

The permanence test in line 15 is triggered by a change of the request instant vector $\overrightarrow{t_{\text{latest}}}$ or the membership vector $\overrightarrow{g}$. The protocol emulation reads the message from the top of the heap of non-permanent messages. If condition (1) fires, the message is removed from the heap of non-permanent and inserted into the heap of permanent pending messages. The retrieval of a message from the top of the heap with the subsequent evaluation of the permanence condition proceeds until the heap becomes empty or condition (1) becomes false. As soon as the permanence test fails for a message, the permanence checking is finished, because the ordering of the heap ensures that the permanence test also fails for all other messages in the heap.

A simulation step in the simulation of the physical CAN network is triggered by the availability of permanent pending messages (see line 23) and a simulation time that is earlier than all elements of the request instant vector $\overrightarrow{t_{\text{latest}}}$. The protocol emulation removes the message m from the top of the heap of permanent pending messages and forwards the m to the front-end (invocation of a message push function of the front-end). Subsequently, the simulation time and the access delays of the pending messages are updated. For this update, we use the transmission duration $d_{\text{transmission}}$ of message m, which depends on the length of m, the identifier type, the bandwidth of the emulated CAN network, and the bit stuffing overhead.

Since the primary keys of messages in H_{perm} have changed (i.e. different access delays), it is necessary to perform a reordering of the heap. In general, multiple messages will now be ordered by the secondary key (message identifier). For these messages, the selection for transmission in the protocol emulation will occur based on the message priority, which corresponds to the media access control strategy of CAN.

V. Validation and Results

We have validated the protocol emulation algorithm in an implementation of a virtual CAN network in the TTA [13] with a physical network running the TTP/C protocol. In the validation framework, we have also employed the MATLAB/Simulink toolbox TRUETIME for

simulating the behavior of a physical CAN network. TRUETIME [14] is a MATLAB/Simulink-based simulation toolbox for real-time control systems. TRUETIME offers a Simulink network block that can be parameterized with the CSMA/CA medium access control protocol. This validation framework has also been employed for measuring the transmission latencies of virtual CAN networks.

We have performed the validation of the protocol emulation algorithm by running test applications in the implementation of the virtual CAN network and the MATLAB/Simulink framework. The test applications have performed transmission requests at predefined instants and included sequence numbers in the broadcasted CAN messages. The comparison of the observed sequence numbers at the receivers has indicated no differences in the temporal message order between the VCN and the MATLAB/Simulink-based simulation of a physical CAN network.

A. Measurement Framework

The measurement framework employs an implementation of virtual CAN networks in the TTA and employs the TTP/C protocol as the physical network. The nodes employed in the prototype implementation use the embedded real-time Linux variant Real-Time Application Interface (RTAI) [15] as their operating system. The RTAI version used for the prototype implementation combines a real-time hardware abstraction layer with a real-time application interface for making Linux suitable for hard real-time applications [16]. The packet service and the CAN controller emulation have been realized as middleware and implemented via Linux kernel modules.

A distributed measurement application uses off-line computed message transmission request tables as inputs for the virtual CAN network. The measurement application at every sender node accesses the virtual CAN network and requests message transmissions at the points in time specified in the request table (see Figure 9). The table also determines the length and identifier of each transmitted message. The data area of the CAN message contains a message index, which uniquely identifies a particular message transmission request along with the node from which the message originated.

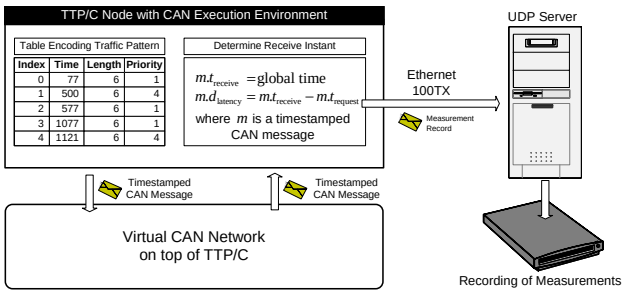

Fig. 9. Measurement Framework

Whenever a message m is received, we assign a timestamp $m.t_{\text{receive}}$ to the received message. Due to the availability of the global timebase of the TTA, we can compute the transmission latency $m.d_{\text{latency}}$ as follows:

$$m.d_{\text{latency}} = m.t_{\text{receive}} - m.t_{\text{request}} \quad (2)$$

where $m.t_{\text{request}}$ denotes the point in time at which the message needs to be sent according to the off-line computed message transmission request tables. The transmission latency $m.d_{\text{latency}}$ incorporates queuing delays of the virtual CAN network at the sender, latencies of the underlying network and execution times of the CAN middleware. A measurement record is constructed with the index of the sending TTP/C node, the sequence number of the message, the receive instant $m.t_{\text{receive}}$, and the transmission latency $m.d_{\text{latency}}$.

$$\langle \text{sender node},\ \text{msg. sequence number},\ m.t_{\text{reception}},\ m.d_{\text{latency}} \rangle$$

This measurement record is stored in a UDP packet and transferred to a workstation that executes a UDP server. The workstation collects the measurement records from TTP/C node and stores them into a dedicated file for each observing node for a later analysis.

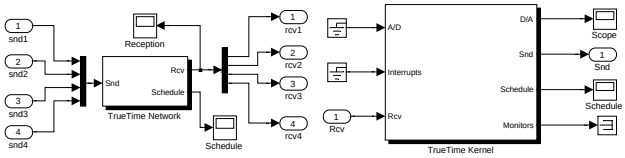

Fig. 10. MATLAB/Simulink Models for Node (left) and Network (right)

B. Simulation Framework

We have employed a framework based on the MATLAB/Simulink toolbox TRUETIME for simulating the behavior of a physical CAN network, when provided with a particular message pattern as input. TRUETIME [17] is a MATLAB/Simulink-based simulation toolbox for real-time control systems. TRUETIME supports the simulation of the temporal behavior of tasks in a host computer, as well the simulation of the timing behavior of communication networks. For this purpose, it offers two Simulink blocks: a computer block and a network block. The blocks are variable-step, discrete, MATLAB S functions written in C++. The computer block S-function simulates a host computer with a flexible real-time kernel and user-defined tasks. Every task is associated with code (e.g., C functions), which is executed during the simulation. The network block operates event-driven and is triggered by messages entering or leaving the network. The network block is parameterized with the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA) and the transmission rate. A send queue is used to hold all messages currently enqueued in the network.

While TRUETIME has primarily been designed for analyzing the effects of timing non-determinism on control performance, the high flexibility of the computer block and the support for the CAN media access control protocol (CSMA/CA) make this tool well-suited for simulating a conventional CAN system. We use the TRUETIME

computer and network blocks (see Figure 10) to model a CAN system consisting of four nodes. Every node employs a TRUETIME computer block that is connected to the network block modeling the CAN bus. In every node, a task is executed that transmits messages according to the message transmission request table as described in the measurement framework. At the points in time specified in this table, the task passes CAN messages to the TRUETIME network. Each CAN message is assigned the priority and length as specified in the table. The data area contains the point in time of the message transmission request and a unique index to identify the message.

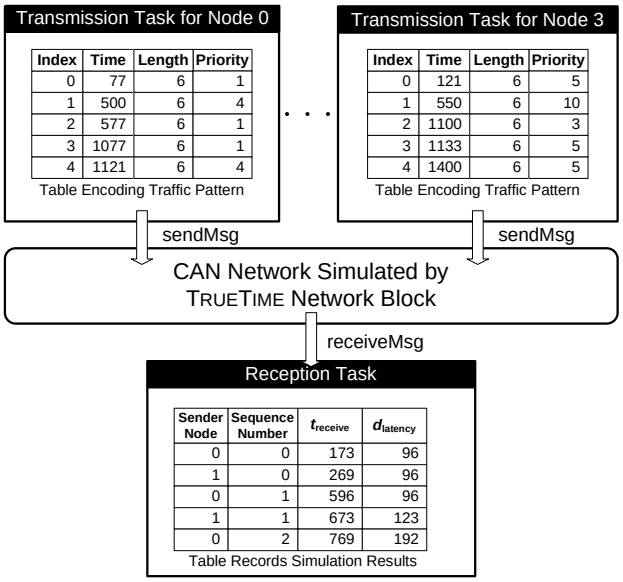

Fig. 11. Simulation Tasks

One of the nodes also hosts a reception task. The reception task is an interrupt handler that is triggered by message receptions. This task retrieves the CAN message from the TRUETIME network and calculates the transmission latency from the timestamp contained in the message and the current simulation time. The determined transmission latency is written into a file for a later analysis. The interplay between the transmission and reception tasks is visualized in Figure 11.

C. Results

For the validation of the protocol emulation algorithm we have used both a communication matrix from a real-world automotive application, as well as synthetic message patterns. The communication matrix from the real-world automotive application originates from a powertrain network and consists of 102 periodic messages. The message periods range from 3.3 ms to 1 s, the number of data bytes is between 2 and 8 bytes. Messages comply with the standard-CAN format [1] and contain 47 control bits, thereby resulting in a total message size between 47 and 111 bits. The overall network bandwidth required for the exchange of these messages is 300 kbps.

The synthetic message patterns consist of CAN messages complying with the extended format [1]. An extended CAN message contains 67 control bits and between 0 and 8 bytes of effective data. Consequently, a complete CAN message has a length between 67 and 131 bits. In addition, there is a data dependent overhead for bit stuffing to prevent more than five consecutive bits of the same polarity.

We have employed two groups of input message sets in the synthetic message pattern. One group consists of sporadic messages, a second one uses aperiodic messages. In both groups, we have used maximum bandwidth consumptions in order to determine the effects of different message loads (e.g., 500 kbps and 1 Mbps) on the protocol emulation.

Using the measurement and simulation frameworks, we have compared the sequence of received messages as indicated by the measurement records in the MATLAB/Simulink simulation of a physical CAN network and the implementation of a virtual CAN network. In test runs without protocol emulation, the message sequences have been different for physical and virtual CAN networks. With enabled protocol emulation, the sequence numbers in the measurement records have shown an identical message order.

VI. CONCLUSION

Based on previous work for the establishment of event channels on top of a time-triggered communication protocol, this paper has presented a protocol emulation algorithm that establishes the temporal message order of a physical CAN network. By offering to legacy applications a communication service that provides the same temporal message order as a physical CAN network – the platform these legacy applications have been developed for – we simplify the reuse of existing CAN-based software. Protocol emulation relieves designers from a costly adaptation of legacy software in order to make them fit for the integration into a time-triggered architecture. An effective solution for the integration of CAN-based legacy applications into a time-triggered architecture is of particular interest to the automotive industry. The CAN protocol is widely used in present day automotive networks for powertrain and body/comfort functions, while time-triggered architectures will be deployed in the safety-critical electronic systems of future by-wire cars.

ACKNOWLEDGMENTS

This work has been supported in part by the European IST project DECOS (IST-511764) and the European IST project ARTIST2 (IST-004527).

REFERENCES

[1] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[2] R. Obermaisser. CAN Emulation in a Time-Triggered Environment. In *Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE)*, volume 1, pages 270–275, 2002.

[3] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms – An Integrated Architecture*. Real-Time Systems Series. Kluwer Academic Publishers, November 2004.

[4] IEC: International Electrotechnical Commission. *IEC 61508-7: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures*, 1999.

[5] E. Bretz. By-wire cars turn the corner. *IEEE Spectrum*, 38(4):68–73, April 2001.

[6] J. Rushby. Bus architectures for safety-critical embedded systems. In *Proc. of the First Workshop on Embedded Software*, 2001.

[7] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan, June 1992.

[8] Intel Corporation. *82527 Serial Communications Controller, Controller Area Network Protocol*, December 1995.

[9] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, May 1994.

[10] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys (CSUR)*, 32(1):12–42, 2000.

[11] R. Obermaisser and P. Peti. Comparison of the temporal performance of physical and virtual can networks. In *Proceedings of the IEEE International Symposium on Industrial Electronics*, Dubrovnik, Croatia, June 2005.

[12] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[13] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[14] J. Eker and A. Cervin. A Matlab toolbox for real-time and control systems co-design. In *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[15] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, April 2000.

[16] RTAI Programming Guide, Version 1.0. Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM), Italy, September 2000. Available at `http://www.rtai.org`.

[17] D. Henriksson, A. Cervin, and K.E. Arzen. Truetime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002. Department of Automatic Control, Lund Institute of Technology.