The Startup Problem in Fault-Tolerant Time-Triggered Communication

Wilfried Steiner
Real-Time Systems Group
Vienna University of Technology, Austria
w.steiner@computer.org

Hermann Kopetz
Real-Time Systems Group
Vienna University of Technology, Austria
hk@vmars.tuwien.ac.at

Abstract

Fault-tolerant time-triggered communication relies on the synchronization of local clocks. The *startup problem* is the problem of reaching a sufficient degree of synchronization after power-on of the system.

The complexity of this problem naturally depends on the system assumptions. The system assumptions in this paper were compiled from cooperation with partners in the automotive and aeronautic industry.

We present a general startup strategy for safety-critical systems that discusses the solution to the startup problem from an abstract point of view. From this abstract view we derive and analyze a new startup algorithm that is used in a TTP/C research derivative protocol (LTTP). We also analyze the FlexRay startup algorithm and discuss its behavior in presence of simple failures. The analyses were done by exhaustive fault simulation using the SAL model checker.

1 Introduction

A fault-tolerant system calls for a distributed solution where the critical tasks are replicated on several "nodes" to tolerate the permanent loss of parts of the system. A major problem in distributed systems is the communication of information between nodes. A straight-forward solution is an individual direct connection of every pair of nodes that have to exchange information, and, in fact, this approach is used in certain modern aeroplanes. An alternative way for message exchange is the usage of shared communication media such as bus or star topologies, which allows a reduction of physical connections.

The application of shared resources, however, introduces a mutual exclusion problem and, hence, dedicated communication protocols have to be used to guarantee that each node connected to the shared medium will get its specified amount of bandwidth for communication. Time-division multiple-access (TDMA) protocols, such as FlexRay [3] or TTP/C [8], which are protocols that off-line specify the access pattern of nodes to the shared medium, are promising due to low transmission latency and transmission jitter. Also, the construction of protection mechanisms that supervise the access of nodes to the medium is straight forward once the system is synchronized. In order to establish synchronization, TDMA protocols have to solve the *startup problem*.

Claesson *et al.* present a strategy for the solution of the startup problem that is based on unique message lengths and full-duplex communication links [2]. Lönn discusses startup algorithms in [11] and formally verifies with Pettersson a particular startup algorithm in [12]. The startup algorithms discussed within this paper, which are startup algorithms

based on unique timeouts, are similar to the startup algorithm of the Token Bus protocol [6]. Krüger introduces such an algorithm for the startup of time-triggered protocols [10]. The TTP/C startup algorithm as specified in [8] is based on this principle.

The presented approaches are based on fault-hypotheses which assume certain restricted fault-models for an individual node, e.g., fail-silence. Despite the research of the startup problem, none of the presented work tolerates the arbitrary failure of one node (e.g. masquerading) [14]. Hence, in further research, [18], we aimed to establish a sufficient degree of fault tolerance (see the fault hypothesis in Section 2.2) by introducing a guardian concept. In this work we tried to avoid changes in the startup algorithm of TTP/C which led to three necessary design choices for the guardian instances:

- the guardian has to execute a "semantic filter", that is, messages are analyzed by the guardian and, if a semantic failure is detected, the message is transformed into a syntactically faulty message, by truncation of the message,
- the centralized guardian instances have to use "interlinks" which are uni-directional direct connections between the two centralized guardians, such that a centralized guardian receives the messages transmitted on the respective other channel, and
- any one non-faulty guardian has to be powered-on before any non-faulty node starts to transmit messages.

This paper presents a general startup strategy that makes these design choices optional by moving fault-tolerance functionality from the guardian instances to the startup algorithm. We start this paper in Section 2 by thoroughly discussing our system assumptions, which include the physical structure, the fault hypothesis, the minimum system requirements, timing assumptions, and the synchronous operation mode and steady state. Section 3 formulates the startup problem. In Section 4 we present the general startup strategy. Section 5 discusses a lower bound on the required nodes/channels to reliably determine when steady state is reached. In Section 6 we discuss two particular startup algorithms: a new LTTP startup algorithm that was developed according the general startup strategy and the FlexRay startup algorithm. The analysis of both algorithms was done by model-checking experiments. This paper concludes in Section 7.

2 System Model

2.1 Physical Structure

Physically, a system, consists of node computers, nodes, and a replicated communication medium, channels. The channels are implemented as half-duplex connections. To avoid medium access by a faulty node we specify guardians that can be either local at the nodes [19], or central at hubs [1]. If central guardians are implemented, these components can be connected via interlinks. Interlinks are uni-directional channels that allow the central guardian of channel X to receive messages of channel Y and vice versa. This paper does not cover the construction of a guardian instance. A general study of central guardians was done in [15, Chapter 5], where two particular implementations are presented also. A system in star topology is depicted in Figure 1.

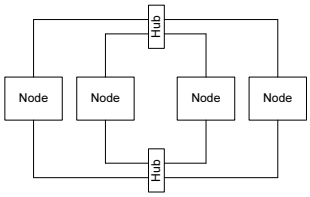

Figure 1: System of four nodes and two channels

2.2 Fault Hypothesis

Each node and each communication channel (together with its guardian) forms one fault-containment region (FCR) [9], that is, those components fail statistically independently. The failure modes of the different FCRs are defined as follows. A node is allowed to fail "arbitrarily", that means it may:

- send arbitrary signals,
- at arbitrary times,
- for arbitrary durations.

The fault model of a channel is defined to be "passive arbitrary". That means a faulty channel/guardian:

- may delay and accelerate a message only for an upper bound in time,
- may relay a received message only to a subset of connected nodes, and
- may not create correct messages.

The system has to tolerate the permanent loss of any one of its FCRs in the defined failure mode. We say the system has to guarantee its services under a "single-failure hypothesis".

In the fault hypothesis we defined the failure mode of a faulty node as arbitrary faulty and for a faulty hub as passive arbitrary faulty. On a logical, algorithmic level that means:

- a faulty node is allowed to send an arbitrary sequence of messages and noise (that is activity that is not recognized as a valid message), with arbitrary intervals of silence in between two accesses to the communication channels.
- a faulty channel may relay valid messages only to a (possibly empty) subset of nodes whenever a node sends a valid message. Furthermore, a faulty channel is allowed to send noise itself to a subset of nodes at any time.

Byzantine Failure We understand a Byzantine-faulty component as a component that yields an asymmetric failure behavior. Different components that interact with the Byzantine-faulty component may see the output of the Byzantine-faulty component inconsistently. In a distributed system, which solely exchanges messages via a shared medium, the impact of a Byzantine-faulty component is limited: the Byzantine-faulty component cannot send different messages to different components (as it would be possible in fully-connected topologies). Still the Byzantine-faulty component may exhibit its asymmetric behavior in the borderland between digital and analog world, since it is impossible to perfectly synchronize the local clocks in a distributed system [13]. Algorithms that are based on the values of the

local clocks are inherently vulnerable to a Byzantine-faulty component. For example: assume a given deadline when a message of a Byzantine-faulty component has to arrive. The Byzantine-faulty component sends the message "just about" this deadline which potentially causes a component A to classify the message as timely accurate while a component B detects a timing failure, since B's local clock is slightly faster than the clock of A. We call such a failure a "temporal Slightly-Off-Specification" (temporal SOS) failure. On the other hand slight changes on the voltage levels at the physical communication lines may cause a component A to decode a message sent by a Byzantine-faulty component differently than a component B. We call such a failure a "value SOS" failure.

SOS failures have been examined during fault-injection experiments and as an architecture decision the central guardians were equipped with control mechanism to transform the asymmetric SOS behavior into detectable symmetric failures. This method is also called "Byzantine filtering" in the literature [4].

This interpretation of a Byzantine faulty component abstracts from failures caused by metastability [7], [21]. A faulty node may send such a metastable signal that can be interpreted by two receivers differently. However, such a signal would have to propagate through the central guardians. By increasing the logic in the central guardians as well as by decreasing their clock speed, the probability of the propagation of the metastable signal can be reduced.

Information Loss Our system model, which is based on a shared communication medium, causes the manifestation of a particular failure behavior: a faulty component may not only introduce additional faulty information, it can potentially cause a reduction of correct information. In our case this reduction can be caused by a faulty channel that drops messages but can also be caused by a faulty node that destroys a message of a correct node by sending at an inappropriate point in time.

2.3 Minimum Configuration

We require a minimum configuration of four nodes to guarantee the correct execution of the clock-synchronization algorithm. This is a natural consequence of the defined fault hypothesis:

- the failure behavior of a faulty node may be arbitrary
- in general we need $3k + 1$ nodes to tolerate k arbitrary faulty nodes
- we allow one node to fail
- for $k = 1$ we need 4 nodes to tolerate one arbitrary faulty component

We also require a minimum number of two channels:

- the failure behavior of a faulty channel may be passive arbitrary
- we allow one channel to fail
- having two channels allows the good channel to mask the failure of the faulty channel

As we will discuss in Section 5, the minimum configuration requirements impose restrictions for the startup algorithm.

2.4 Timing Assumptions

The system model within this paper is similar to the synchronous system definition given by Verissimo [20, Chapter 3]. We call our system model "eventually synchronous". It is defined by the following timing bounds:

1. Bounded Propagation Delay: there is a known upper bound δ^{pd} on the propagation delay. δ^{pd} consists of the time it takes for sending the first bit of a message, transporting, and receiving the first bit of a message over a communication medium.

2. Bounded Clock Drift: every node n has a local clock C_n with a known bounded rate of drift $\rho_n \geq 0$ with respect to physical time.

3. Bounded Processing Time: there are known upper and lower bounds on the time required by a process to execute a processing step.

4. Uncertain Power-On Time: the time, $\Delta_{power-on}$, that a node/channel needs until it is able to participate in the distributed system, is bounded but not known.

The first three items are natural assumptions and usually reflect the upper bounds that are required for real-time systems.

Item 4 in this enumeration is the most critical one because it makes the distinction of a crashed component from a late component impossible; this follows from the impossibility result on crash failure detection in asynchronous systems [5].

Of course, the easiest way to circumvent the problem with item 4 is to ignore it and require known upper bounds for all actions in the computer system, that is, to move to a fully synchronous system. However, the power-on line is external to the communication protocol. By using the eventually synchronous system model, the developed algorithms apply as well to those applications that require bounded power-on times at design time, and those that do not. And, of course, less assumptions increase the assumption coverage of the system model.

2.5 Synchronous Operation and Steady State

During synchronous operation the nodes execute a time-triggered communication (also called a time-division media-access) strategy. Such a strategy splits up time into (non-overlapping) pieces of not necessarily equal durations τ^{slot}, which are called *slots*. Slots are grouped into sequences called *TDMA rounds* of length τ^{round}. The sequence of slots is the same for all TDMA rounds. An example of a TDMA round with four slots is given in Figure 2.

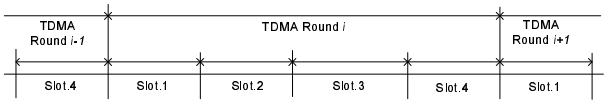

Figure 2: TDMA round of four slots

The knowledge which node occupies which slot in a TDMA round is static, available to all components a priori, and equal for all TDMA rounds, we call this information the "TDMA round layout". We assume that each node will occupy only one slot per TDMA round.

Steady state is then reached when the synchronous operation tolerates the failure of a component according the fault hypothesis.

3 Problem Specification

There are two key properties that a startup algorithm has to guarantee: timely startup and safe startup.

Property 1 *Timely Startup:*
Whenever at least a minimum configuration of nodes and channels is powered-on, a startup algorithm establishes synchronous communication of the non-faulty components within an upper bound in time.

The minimum configuration is specified in Section 2.3. The "whenever" factor in the property specification is highly important, since it does not specify an upper bound in time until a minimum configuration is powered-up. Note also that this timeliness property is stronger than a "liveness" property: in contrast to liveness properties, timeliness properties require a known upper bound on the duration after which a property has to be established.

Property 2 *Safe Startup:*
When the startup algorithm terminates, all correct nodes that communicate synchronously are synchronized to each other.

The safety property ensures that the startup algorithm will not produce multiple "cliques", that are sets of nodes that communicate synchronously within the set but not with nodes in other sets. However, cliques can be formed temporally during the startup process.

Ensuring these properties is a non-trivial problem, even in the failure-free case, since there is a cyclic dependency between the synchronization process and the communication process:

- each synchronization process requires the timely and deterministic delivery of messages, and
- a shared broadcast medium requires synchronization in order to timely and deterministically deliver messages.

It is not possible to address these two points in isolation. Solutions, as for example "leader-election" algorithms are not feasible, since they rely on message exchange and message exchange relies on synchronization. A startup algorithm has to address both issues at the same time.

4 General Startup Strategy

The general strategy for starting up the system is depicted in Figure 3. It identifies three protocol phases: integration, coldstart, and sync.

After power-on (that is after the node is initialized) the node starts the integration phase. As defined by the system model, each slot in the communication schedule is a priori assigned to a sending node and each message carries the identifier of its sender. Hence, the node listens to the communication channels and has to identify, based on the messages received, if there is a sufficient number of nodes communicating synchronously. If such a set exists, the node

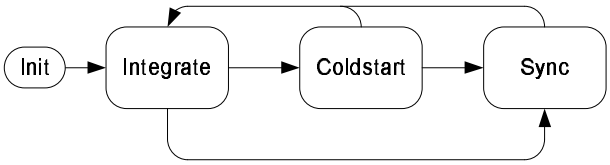

Figure 3: General startup strategy

integrates into this set and becomes synchronized. If such a sufficient set does not exist, the node enters the coldstart phase.

In the coldstart phase, the node waits for coldstart signals, that are used to indicate a particular position in the TDMA round schedule. A node that receives a coldstart signal will set its local state to this position and reply in its assigned slot. If a node does not receive a coldstart signal for a specified timeout, it sends such a signal (only a subset of nodes may be configured to send a coldstart signal). The nodes are able to acquire the number of nodes that react to the coldstart signal by counting the replies. The coldstart phase ends when a sufficient set of nodes has been synchronized. In general it may also happen that only a subset of nodes in coldstart will reach synchronous operation. For this reason each startup algorithm should define conditions for a re-transition to the integration phase, such that nodes that did not reach the sync phase are able to integrate to the established synchronous communication.

In the sync phase the node has reached synchronous operation (but not necessarily steady state). If synchronization is lost, the nodes restart the startup process with the integration phase.

The transitions between the different phases of the startup strategy can be taken either by the expiration of timeouts or by the reception of a sufficiently long sequence of messages per TDMA round. It is highly important that a faulty node or channel cannot be able to spread such a sequence of messages (e.g. by masquerading a number of different nodes) that will cause a non-faulty node to take an incorrect transition between startup phases.

4.1 Integration

During sync phase each node sends a message which carries its sender's identifier in its a priori assigned sending slot. Integration is, hence, a straight forward task: when a node receives a sufficiently long sequence of messages during one TDMA round, the node knows that synchronous communication exists, and, furthermore, is able to adjust its local state to the state contained in the received messages. The necessary length of the sequence has to be longer than the sequence of messages that a faulty node can insert into the distributed algorithm execution. Insert means in this context that a message from a faulty node is relayed by the communication medium. Guardians are able to block faulty messages, which means that not each message generated by a faulty node will be inserted into the distributed algorithm execution.

As defined in our system model, only one slot per TDMA round is assigned to each node and the round layout is equal for each round. Given a guardian that guarantees that a node will only send in its sending slot during synchronous operation, it is guaranteed that a faulty node can insert only one faulty message per TDMA round per channel. The number of messages that an integrating node needs to receive depends on the filtering techniques used in the guardian. An integrating node needs:

- 1-out-of-2 corresponding messages (that is a sequence of 1 message), if the faulty message is detectably faulty

(following the $k+1$ rule), or

- 2-out-of-3 corresponding messages (that is a sequence of 2 messages), if the faulty message is not detectably faulty (following the $2k+1$ rule).

There is an interdependency between the slot to node relation, the required filtering techniques in the guardian, and the required number of messages necessary for integration: if the system model is weakened with respect to the relation of slots to nodes such that a node is allowed to acquire more than one slot per round, a faulty node can simulate a number of nodes which is equal to the maximum number of slots in which it is allowed to send. If we do not use further filtering techniques we have to implement a majority voting: a node has to receive $(k+1)$-out-of-$(2k+1)$ corresponding messages, where k is the maximum number of slots assigned to a node in a TDMA round. However, that does not mean that we require $(k+1)$ distinct nodes, as also correct nodes can be assigned more than one slot per round.

The integration phase can terminate "successfully" or "unsuccessfully": when the node has received a sufficiently long sequence of messages it is able to synchronize to a running system and the integration terminates successfully. If the node is not able to integrate for a given duration, the node terminates the integration phase unsuccessfully and transits to the coldstart phase.

4.2 Coldstart

In the coldstart phase a node a) waits for coldstart signals and b) sends coldstart signals by itself. The system can be configured such that only a dedicated set of nodes, the "core system" is allowed to enter coldstart phase. Nodes that are not in the core system will enter sync phase only by integration. Coldstart signals are starting signals for the nodes to start synchronized communication. There are several ways to construct such a coldstart signal:

- Noise: any kind of activity that can be detected by nodes. This form of coldstart signal highly depends on the physical layer used for the realization of the communication channels.
- Semantic-Free Coldstart Message: a valid unique message that is sent as coldstart signal. The reception of the message alone is evaluated as coldstart signal.
- Semantic-Full Coldstart Message: a valid unique message that is sent as coldstart signal. This coldstart signal carries additional information, for example where to start in the communication schedule.

In a fault-tolerant system it is necessary to configure more than one node to send a coldstart signal and, hence, there is always the possibility that these nodes send their coldstart signals at approximately the same point in time causing a contention. According to our system model the communication channels are half-duplex and, hence, contentions cannot be detected immediately by the senders. Furthermore, propagation delays on the communication channels and deaf windows in the nodes (that are phases when a node switches from receiving to transmitting) make it impossible that a node receives the coldstart signal from another node instantaneously. If the contention is not resolved, the quality of the initial synchronization depends on these parameters (propagation delay and deaf window). A contention-resolving algorithm can be used to ensure that eventually there is only one node that sends a coldstart signal that will not result in a contention. A contention scenario with two nodes is depicted in Figure 4.

During the listen periods, the nodes try to receive coldstart signals. At some point in time the nodes decide that they have to initiate the coldstart themselves. During the following pre-send period the nodes are not able to receive

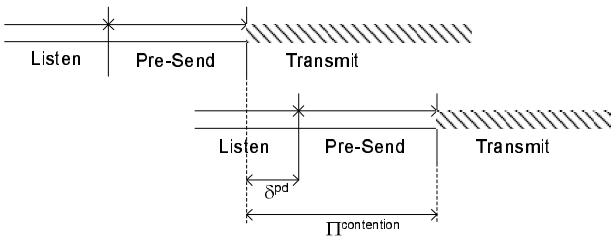

Figure 4: Initial precision after a contention

messages from the communication channels. Finally, the nodes broadcast their messages. We see that the quality of the initial precision $\Pi^{contention}$ depends on the maximum propagation delay δ^{pd} (the propagation delay is assumed also to cover digitalization errors at the receiver) and the duration of the pre-send period $\Delta^{pre-send}$.

$$\Pi^{contention} = \delta^{pd} + \Delta^{pre-send} \tag{1}$$

We can distinguish two types of contentions depending on the topology of the communication medium.

- A *physical* contention occurs, when two or more nodes send at approximately the same time and the signals of these nodes physically overlay on the medium. This type of contention may occur in a bus topology.
- A *logical* contention is a result of the replication of the shared medium where the replicas are controlled by mutually independent instances, e.g. central guardians. Each of these instances guarantees a transmission free of physical contentions on one replica. However, since these instances are independent of each other, nodes that start to broadcast at approximately the same time may occupy only a subset of the replicas each. A receiver, therefore, will receive messages from different senders on the replicas of the communication medium. Logical contentions may occur in a star configuration of a system.

This paper focuses on the star topology and therefore on logical contentions.

A contention resolving algorithm, guarantees that there exists an upper bound in time, when the access of at least one node will not result in a contention:

Property 3 *If several nodes have produced a contention at their n-th access to the shared medium, there exists an x such that the $(n + x)$-th access of at least one node in this set will not result in a contention.*

This property is essential for the coldstart phase since it guarantees that even if there are more nodes sending their coldstart signal at approximately the same point in time, there exists an upper bound in time when one node will send its coldstart signal without a contention.

The contention problem naturally arises in communication networks based on a shared communication medium and, hence, communication protocols have to provide solutions for this problem. A summary of contention resolving algorithms of well-established communication protocols (such as Avionics Full-Duplex Ethernet, Token Bus, etc.) can be found in [15, p.35 ff.]. Such protocols usually use priority-based algorithms where the priorities are realized as different unique timeouts; e.g. a re-try timeout – if a contention occurs the one node with the shortest timeout will be

the first to re-transmit its message.

The coldstart will terminate "successfully", if the node finds a sufficient number of nodes/channels synchronized. The node then transits to the sync phase. The coldstart terminates "unsuccessfully" if the number of nodes/channels is not sufficient. When the coldstart phase terminates unsuccessfully another coldstart phase may be started immediately or the integration phase may be re-started, which allows the node to check wether a different set of nodes reached sync phase already. Determining, whether a set of nodes/channels is sufficient, or not, is done at design time and can be bound to the following numbers:

1. the numbers of nodes/channels necessary to guarantee the correct operation of the application, or
2. the numbers of nodes/channels necessary to allow an unsynchronized node to integrate into a synchronized system.

In this paper we are forced by the minimum configuration requirements to select the numbers according Point 2. The discussion of this design space restriction is done in the following section.

5 Impossibility of Steady State Detection

Lemma 1 *Although a set of nodes reach steady state, it is not possible to design an algorithm that detects when steady state is reached under the given system model.*

Discussion: The discussion is based on the minimum number of nodes, an analogous discussion can be done regarding the minimum number of channels.

As defined in Section 2.5, steady state is then reached when a failure according the fault hypothesis can be tolerated. In our system model that requires at least three non-faulty nodes to operate in steady state. Each node is able to count the number of nodes that reached synchronous operation by the messages it receives, due to the bijective relation of slots to nodes.

A steady state detection algorithm requires the reception of messages from at least three *other* nodes: only then it is guaranteed that there are three non-faulty nodes in steady state, as one of the four nodes may fail.

However, the faulty node may be fail-silent from the beginning and never send a message. As our system model specifies a minimum of four nodes, only the remaining three non-faulty nodes will exchange messages. Each node will therefore only receive messages from two other nodes. Hence, although steady state is reached, the detection algorithm fails.

Due to the unknown time bound (Point 4 in Section 2.4) we cannot specify a timeout to classify a node that never sends a message as faulty as we cannot distinguish such a faulty node from a node that is powered on relatively late to the other nodes. Basically, we face the same problem as Fischer, Lynch, and Paterson [5] where a fail-silent node cannot be distinguished from a "slow node".

The generalization from four nodes and one faulty node to n nodes and k failures is straight forward:

- the number of components to tolerate k faulty components is a function $f(k)$ of its failure behavior (in the case sketched above we want to tolerate one faulty node that may be arbitrarily faulty: $f(k) = 3k + 1 = 4$),

- since each faulty component may fail silent for an arbitrary number of intervals during algorithm execution, $f(k) + k$ components are required to reliably determine the point in time from which on the specified failure of a component is tolerated (in the case sketched above $f(k) + k = 5$),
- the overall number of nodes, n, has to satisfy: $n \geq (f(k) + k)$ (which is violated in the previous special case $(4 \geq 5) = false$)

We conclude that it is impossible to construct a detection algorithm that correctly detects steady state and that terminates under the given system model. ■

We identify the following three ways to overcome Lemma 1:

- Change timing assumptions: we may add an additional requirement to our system model: the knowledge of the worst case power-on time of nodes/channels.
- Change minimum configuration assumptions: we may require a higher number of nodes/channels for a minimum configuration.
- An iterative startup process, where nodes may transit from sync phase back to integration (depicted in Figure 3 as transition from sync to integrate phase).

As we are bound by our system model we have to accept an iterative startup process. As it follows from the impossibility result there exist scenarios in which an already synchronized system will loose the established synchronization. Theses scenarios take place when a faulty node or channel behaves correctly during the startup phase while a sufficient number of non-faulty nodes/channels is not powered-on (and, hence, not able to synchronize). Before the necessary non-faulty nodes/channels integrate into the synchronous communication the faulty component causes the system to lose synchronization. However, as the power-on times of all non-faulty components are bound (but not known) there exists a point in time when sufficient components are powered on and the system will stay synchronized.

6 Assessment of the LTTP and the FlexRay Startup Algorithms

This section discusses the startup algorithms of LTTP (Layered TTP) and FlexRay. Both algorithms have been analyzed using the SAL model checker developed by SRI International. Due to space limitations we only outline the algorithms, detailed descriptions can be found in Steiner's PhD thesis [15][1] and the FlexRay specification [3].

6.1 LTTP

Layered TTP is a research derivative of TTP/C. Its prime purpose is the layering of TTP/C services such that it can be easily tailored to a customer's needs. With respect to startup LTTP tries on the one hand to minimize the required guardian functionality by using mechanisms like majority voting and on the other hand to decouple the guardian functionality from the protocol to enhance diversity and minimize the possibility of common mode failures.

[1] Due to an ongoing IP process the thesis is currently not published, but will be published at conference time.

6.1.1 Algorithm Overview

The startup algorithm of LTTP was developed with respect to the general startup strategy discussed in Section 4 and with heavy support of model-checking experiments. It is fully compliant to the system model discussed in Section 2.

The algorithm uses the following message types:

cs-frames: cs-frames are the starting signals in the system and have length d^{cs}. cs-frames only carry the binary information that they are cs-frames (semantic-less coldstart messages). A successful cs-frame will signal the start of a TDMA round.

ack-frames: acknowledgment frames are used to acknowledge a received cs-frame. ack-frames have length d^{ack}, where $d^{cs} = d^{ack}$. ack-frames carry the current slot position in the TDMA schedule.

i-frames: i-frames are the regular frames transmitted when a set of nodes is synchronized. i-frames have a minimum length of d^i, where $d^{ack} < d^i$. i-frames carry the current slot position in the TDMA schedule.

cu-frames: cu-frames (cleanup-frames) are used for the speedup of the startup algorithm. cu-frames have length d^{cu}, where $d^{ack} < d^{cu} < d^i$.

The startup algorithm is represented by the state machine in Figure 5.

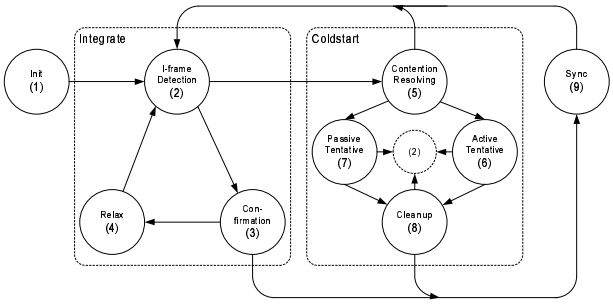

Figure 5: LTTP startup algorithm - without speedup

Init(1): In (1) the node performs its internal initialization and is not able to receive messages from the channels. When a node finishes its internal initialization it transits to (2).

i-frame Detection(2): During (2) the node tries to receive valid messages for a duration of one TDMA round. If it receives a cs-frame it re-enters (2), which basically means, that the timer is reset. By doing so, it is guaranteed that the node will not accept a cs-frame earlier than one TDMA round after a previously sent cs-frame. If an i-frame is received, the node synchronizes on the contained state information and transits to (3). If nothing is received for a duration of τ^{round}, the node transits to (5).

Confirmation(3): Here a node waits for a second i-frame with corresponding state information to the previous i-frame; this is called tentative integration. If during one TDMA round (which is started with the reception of the first i-frame in (2)) no such i-frame is received the node transits to (4). If during this duration an appropriate i-frame is received, the node transits to (9). This confirmation can be generalized in a way such that more than one i-frame has to be received for a valid state confirmation. However, to tolerate a single failure one i-frame suffices for confirmation.

Relax(4): The unsuccessful confirmation in (3) can be a result of the synchronization to a faulty message in (2): the first frame the integrating node received was sent from a faulty node and contains a faulty state. Since the node synchronizes to this faulty state, following correct messages will not confirm the integrating node. The purpose of the Relax(4) state is, to guarantee that the integrating node will not again synchronize to the message of the same, potentially faulty, sender. Hence, the node waits in relax state for a sufficiently long duration d^{relax} and does not react to traffic on the channels before it transits back to (2). For simplicity we assume here that $d^{relax} = \tau^{slot}$.

Contention Resolving(5): In the Contention Resolving(5) state a node sets its local timer to $\tau^{long-startup}$ timeunits and waits for the reception of messages:

- if an i-frame is received, the node transits back to (2).
- if a cs-frame is received the node sets its IFC counters, that are channel **I**ndividual **F**rame **C**ounters, accordingly (that means channel individually to 1 if a cs-frame has been received on the respective channel). The node then checks, whether a cs-frame is also received on the respective other channel for $\varphi_{max}^{inter-channel}$ timeunits. A message broadcasted by a node on both channels at the same point in time will, in general, arrive at the receiver at different points in time, as the propagation delays on the channels may be different. $\varphi_{max}^{inter-channel}$ is the maximum inter-channel jitter. If a cs-frame is received on the other channel as well, the corresponding IFC is also set to 1. The node transits to (6) afterwards.
- if for $\tau_i^{long-startup}$ no frame is received, the node sends a cs-frame itself and enters (7).

Upon transition to (6) or to (7) the node also sets its local timing to the start instant of the cs-frame received on the faster channel, corrected by the propagation delay, or to the start instant of the cs-frame transmission.

Our model-checking studies showed that the startup timeout $\tau^{long-startup}$ should be configured to:

$$\tau_i^{long-startup} = 3 * \tau^{round} + \tau_i^{startup} \tag{2}$$

where τ^{round} is the duration of one TDMA round and $\tau^{startup}$ is a node-unique timeout, specified as:

$$\tau_i^{startup} = \begin{cases} 0 & i = 0 \\ \sum_{j=1}^{i} \tau_{j-1}^{slot} & i > 0 \end{cases} \tag{3}$$

Note here, that the coldstart period $\tau^{coldstart}$ (which is the minimum duration between two successive coldstart attempts of a node) has to take also the first additional tentative round and the round during integration phase into account. Hence the coldstart periods are given in by:

$$\tau_i^{coldstart} = 2 * \tau^{round} + \tau_i^{long-startup} \tag{4}$$

This algorithm does not specify dedicated listen timeouts to prevent that recently powered-on nodes cause additional contentions with previous coldstarters (that are nodes that sent a cs-frame). Such timeouts are not needed in a system that:

- is in star topology: only logical contentions may occur
- uses semantic-less coldstart messages: in case of a logical contention, a receiving node will still receive two equal cs-frames

In systems that are realized in bus-structure dedicated listen timeout should be specified: the first time a node enters the coldstart phase will cause the node to wait a listen timeout that is longer than the maximum coldstart timeout.

The unique timeouts implement a contention resolving algorithm, which is necessary to guarantee that there are eventually enough nodes that acknowledge the coldstarter.

Active Tentative(6): Here the node executes one tentative TDMA round starting with the first slot in the TDMA schedule. Since one physically faulty component may present itself as two logically faulty components, by alternately sending messages on only one channel, the messages have to be counted on a per channel basis using the IFC. When a node receives an ack-frame that corresponds to its local view on the current protocol state (that means that the slot position set in the received frame is equal to the receiver's current slot position in the TDMA round layout), it increments the respective IFC counter(s) by one. If the node reaches its sending slot in the schedule, the node sends an ack-frame itself on those channels on which it has received the cs-frame in state (5), and increments the respective IFC counter(s). When the tentative round is finished, the node checks whether any of its IFC counters reached the $minimum\_sync$ value. If so, the node transits to (8). If none of the IFC counters has reached the necessary threshold, the node resets its counters and re-enters (2).

The $minimum\_sync$ value depends on the number of nodes that are allowed to execute the coldstart phase (the core system), say n:

$$minimum\_sync = \left\lceil \frac{n}{2} \right\rceil + 1 \tag{5}$$

The discussion of this value is done in [17], where we explicitly address a clique resolving algorithm. The Active Tentative(6) state implicitly incorporates such a functionality in order to guarantee that if there exists a non-faulty node in Sync(9) state, there always exists one other non-faulty node in Sync(9) state as well. Hence, the non-faulty nodes are always in majority as only one faulty node may exist. Our experiments used a core system of four nodes ($minimum\_sync = 3$).

Passive Tentative(7): This state is equal to state (6) with the exception that a node will not send in its sending slot and not increase any IFC counter in its sending slot.

Cleanup(8): The cleanup state is a second tentative round in which i-frames are broadcasted (if the algorithm speedup is used, cu-frames are broadcasted). At the end of this round, each node checks if there are at least $minimum\_sync - 1$ nodes communicating synchronously, to tolerate a fail-silent faulty node that was active during the first tentative round.

Sync(9): This is the synchronous operation of the system. A node cyclically executes the TDMA schedule.

As discussed in Section 5, scenarios are possible in which nodes loose synchronization, e.g. if a system of only two correct nodes and one faulty node are running in synchronous operation. The loss of synchronization detected by continuously monitoring the number of frames received from core nodes. The node executes a watchdog algorithm: upon entering the sync state a node sets its timer to $\kappa^{unstable}$ rounds. Whenever a node detects $minimum\_sync$ nodes in steady state, it feeds the watchdog timer. If the watchdog timer elapses, the node restarts.

6.1.2 Model-Checking Results

Parts of the model-checking results are shown in Figure 6. The presented results are based on a system of four nodes and two channels. The first column depicts the nodes that are configured to send cs-frames (however, all four nodes execute coldstart phase). The second column names the faulty node. The faulty node exhibits an arbitrary failure. Columns three and four show the result of the model-checking run and its execution time for the safe startup property, columns five and six show these values for the timely startup property. Column seven shows the worst-case startup times derived from counterexamples of the model-checking runs. Strategies for tolerating a faulty channel are given in [15, Sec. 6.6.4]

The model-checking experiments have been done similar to the approach presented in [18] where we presented "exhaustive fault simulation" as measure for design and verification of faul-tolerant algorithms: a faulty component is allowed to produce any of the protocol relevant messages (syntactically and semantically correct or incorrect) at any point in time of the model execution. As depicted the LTTP startup algorithm has the desired timeliness and safety properties. We used the model checker in the design loop of the algorithm. Some interesting failure scenarios that popped-up during the design stages of the algorithm as well as detailed descriptions of the worst-case startup times (wcsup) are given in [15, Sec. 6.6.4].

		Safe Startup		*Timely Startup*		
coldstarters ID	faulty ID	eval.	cpu time (sec)	eval.	cpu time (sec)	wcsup (rounds+slots)
1,3,4	1	true	1124.58	true	1565.13	38+3
1,3,4	3	true	801.63	true	2172.89	27+3
1,3,4	4	true	1124.58	true	7638.99	29+2

Figure 6: LTTP – model-checking results for safe startup (property 1) and timely startup (property 2)

6.1.3 LTTP startup speedup

The presented startup algorithm is costly in terms of TDMA rounds. Hence, we propose to use a dedicated TDMA round layout, a "core TDMA round layout", that consists only of a limited number of slots (possibly only four) during the coldstart phase. During the integration phase and the sync phase, the node uses the "user TDMA round layout" which is the application-specific schedule. This speedup requires minor changes in the algorithm's state machine: once a node enters coldstart phase it will stay in this phase until it either receives a sufficient number of cu-frames per a given interval, or a specified number of unsuccessful coldstart attempts has been reached. CU-frames are broadcasted in Cleanup(8) state instead of i-frames.

Exhaustive model-checking studies also showed the correctness of the speedup approach.

6.2 FlexRay

FlexRay is intended to serve as a general purpose automotive communication protocol that addresses ultra-high dependable applications as well. It is intended for the usage of so called x-by-wire applications in the automotive area, e.g. break-by-wire or steer-by-wire applications. Although the assessment in this paper is based on the FlexRay 2.0 specification, the problems found are still open in the FlexRay 2.1 specification.

6.2.1 Algorithm Overview

The FlexRay system model deviates from the system model in this paper in the following ways.

- FlexRay requires only three nodes as minimum configuration for successful startup: FlexRay is forced to use a 1-out-of-2 integration strategy.
- FlexRay allows a node to occupy more than one slot per round: FlexRay requires a semantic filter in guardian instances.
- FlexRay assumes a fully synchronous system: the absolute time until a node is able to participate in startup is assumed to be given.
- FlexRay does not specify a fault hypothesis: for x-by-wire applications we have to assume the fault hypothesis presented in this paper. However, as we will show, even simple failure scenarios are not tolerated.

The FlexRay startup algorithm is depicted in Figure 7. The figure represents states as boxes, arrows are used to depict state-transitions during algorithm execution. Circles represent successful startup termination.

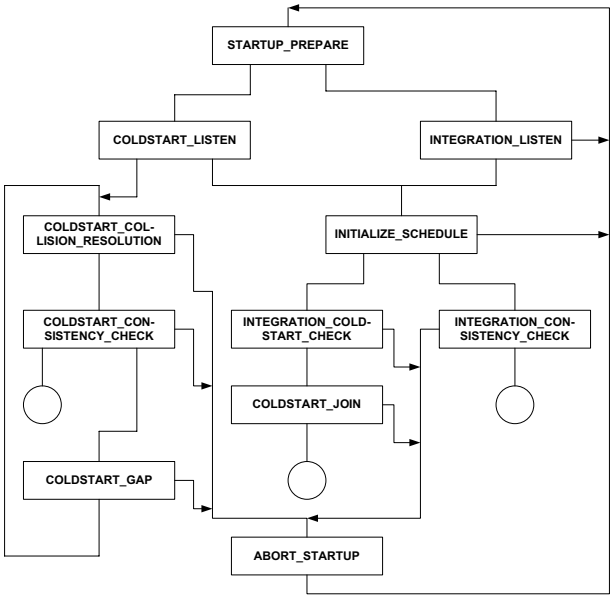

Figure 7: FlexRay startup algorithm (taken from the FlexRay 2.0 specification [3, p. 145])

FlexRay defines dedicated “coldstart” nodes, which are nodes that are allowed to execute the coldstart. Nodes that are not coldstart nodes will not participate in the coldstart process but will only integrate after a sufficient set

of coldstart nodes established synchronous operation. Between the coldstart nodes we can distinguish the "leading coldstart node" and the "following coldstart nodes". This distinction is made dynamically during the execution of the startup algorithm: after power-on, the nodes execute a "wakeup algorithm" whose purpose is to ensure that all non-faulty nodes are ready to execute the startup algorithm. The nodes enter then **COLDSTART LISTEN** where they try to receive FlexRay messages for at least two TDMA rounds. If a "collision avoidance symbol (CAS)", which is a low pulse of a specified duration, or a correct message header is received (but no correct message) the timer is reset and **COLDSTART LISTEN** is reentered. If the message is correct, the node transits to **INITIALIZE SCHEDULE** and tries to integrate. If two TDMA rounds pass by without the reception of either CAS or a correct header, the node transits from **COLDSTART LISTEN** to **COLDSTART COLLISION RESOLUTION**. By doing so it transmits one CAS. A specified timeout after this transition the node starts the TDMA schedule at the first slot and will send in its a priori assigned slot, except it leaves the state before. In the case of a collision more than one node take this transition at approximately the same time, **COLDSTART COLLISION RESOLUTION** shall ensure that only one node remains in this state. The position in the TDMA schedule is used as a priority mechanism: in a well-configured FlexRay system the foremost node in the schedule will be first to transmit a message, all other nodes in **COLDSTART COLLISION RESOLUTION** receive this frame and transit to **COLDSTART LISTEN**. The node that remains in **COLDSTART COLLISION RESOLUTION** is then the leading coldstart node. After four rounds in **COLDSTART COLLISION RESOLUTION**, the leading coldstart node transits to **COLDSTART CONSISTENCY CHECK** and checks whether there exist nodes that used its messages for integration. If there exists at least one such node, it tries to keep up communication for one more TDMA round until it finally concludes synchronized operation reached. If the leading coldstart node does not become acknowledged, it enters **COLDSTART GAP** where it waits for one TDMA round. Again, if CAS or a correct header is received, the node reenters **COLDSTART LISTEN**. After one TDMA round the leading coldstart node reenters **COLDSTART COLLISION RESOLUTION** and tries again to synchronize other nodes, this time, however the node does not broadcast a CAS, but instantly starts with the execution of the TDMA schedule in the first slot. For successful integration of the following coldstart nodes four messages of the leading coldstart node (one per TDMA round) are required. That means that each coldstart attempt consists of four rounds. In addition to this algorithm, each node has a counter that counts the transitions to **COLDSTART COLLISION RESOLUTION**; if this counter reaches a specified upper bound, the node will stop trying to become leading coldstart node and remains in **COLDSTART LISTEN**.

6.2.2 Model-Checking Results

We focused our model-checking studies on the coldstart nodes only as it is a prerequisite that these nodes reach synchronous operation before other non-coldstart nodes will be able to integrate. The model-checking studies found and confirmed startup scenarios where even simple failure modes of a node, e.g. a repeating fail-silence faulty node, prevent a successful system startup. We discuss three startup scenarios next (the corresponding SAL sources and diagrams can be found in [16]).

Fault-Free Startup Scenario: We define the worst-case startup time as the duration starting when at least two nodes leave **STARTUP PREPARE** and ending when all non-faulty nodes reach **SYNC**. The worst-case startup time in the fault-free case was computed to be approximately 15 TDMA rounds. This is a collision scenario, which means all nodes transit to **COLDSTART COLLISION RESOLUTION** at approximately the same time. The leading coldstart node becomes selected and all other coldstart nodes transit back to **COLDSTART LISTEN**. As the nodes need four

messages for integration and the first message from the leading coldstart node is only used for the state transition back to **COLDSTART LISTEN**, this coldstart attempt is lost. The next coldstart attempt of the leading coldstart node is successful.

Resetting Leading Coldstart Node Scenario: As the coldstart algorithm is a priority based algorithm (as it follows from the unique timeouts), there exists one node with the highest priority, which is the node with the foremost slot in the TDMA schedule. This node may exhibit a failure mode such that it always becomes leading coldstart node but fails to send messages for four consecutive rounds, say it is resetting after the third round. It is questionable if the node-internal coldstart attempts counter is a sufficient measure to handle that scenario, as there is plenty of time for the node to completely restart, and hence, resets this counter as well. As a consequence this repeated fail-silence behavior of the node prevents the system from reaching synchronous operation. Such a failure scenario can only be handled by using an appropriate guardian instance. We designed a central guardian that tolerates this failure scenario and leads to a worst-case startup time of $27rounds + 2slots$.

CAS-Babbling Channel: The CAS has a trivial structure: a low-pulse of a specified duration. The assumption that a channel, which may comprise guardian instances, is not able to produce such a signal by its own can not be guaranteed. Hence, we do have to assume that a faulty channel will continually send CAS. As a consequence, the nodes may never leave **COLDSTART LISTEN**, as the reception of CAS on one channel will cause a node to reset its timer in this state.

In general this failure class cannot be solved in a two-channels system. However, the probability that a channel is able to forge a message decreases with the message's complexity. Our proposal requires two changes in the algorithm: a) the node ignores all CAS symbols and uses only messages, which are less likely to be forged by a channel, and b) we have to specify unique timeouts to realize an appropriate contention resolving algorithm. Using this strategy causes a worst-case startup time of $23rounds$.

Figure 8 summarizes the results of a preliminary model-checking study of the FlexRay startup algorithm. Further studies also found scenarios where a faulty node caused the formation of cliques, that is the safe startup property is violated.

Scenario	Safe Startup	Timely Startup
Fault-Free Startup	TRUE	$15rounds + 1slot$
Resetting Leading Coldstart Node	—	∞
Resetting Leading Coldstart Node and a Central Guardian	TRUE	$27rounds + 2slots$
CAS-Babbling Channel	—	∞
CAS-Babbling Channel and CAS-Ignoring Nodes	—	∞
CAS-Babbling Channel, CAS-Ignoring Nodes, and Unique Timeouts	TRUE	$23rounds$

Figure 8: FlexRay – model-checking results for safe startup (property 1) and timely startup (property 2)

6.3 Guardian Functionality

Each system that uses a shared communication medium requires some form of guardian functionality to prevent an arbitrarily faulty node to permanently block the usage of the shared resource by non-faulty nodes. During the synchronous operation of time-triggered communication such a guardian functionality is easy to implement – the

guardian has to grant access to the shared resource to the only one node that is allowed to broadcast at a particular interval and the schedule information is specified off-line. However, as during startup the nodes and the guardians are not synchronized yet, the guardian functionality in this phase is more complex and depends on the startup algorithm used.

One, if not the, major benefit of the LTTP startup algorithm is, that it allows to use a simple leaky bucket algorithm during startup. This means basically, a node that was not silent, e.g. it sent a coldstart signal, is blocked by the guardian for an a priori defined interval. Only after this interval the node is allowed to broadcast again. The leaky bucket algorithm transforms the purely priority-based coldstart algorithm (which is based on unique timeouts) to a "fair" algorithm – the blocking interval has to be specified to be at least as long as to guarantee that a node that broadcasted a coldstart signal will be blocked at the next scheduled point in time of a coldstart message transmission.

From our model-checking experience with the FlexRay startup algorithm we conclude that such a simple solution will not be applicable. Even the guardian used to tolerate the simple failure scenarios presented above has to be tightly coupled to the startup algorithm, to guarantee that each node will only send according its specified timeout-schedule. Also semantic filters have to be used to transform a semantically faulty message to a syntactically faulty message. In LTTP such filtering can be avoided as the usage of different message lengths makes it possible to implement a temporal filter in the guardian: while the guardian is not synchronized only short messages will pass through, long messages will be cut-off and, hence, syntactically invalidated. As the analyzed failure scenarios assumed very simple failures, we conclude that a final guardian instance for FlexRay, that tolerates the arbitrary failure of a node, will result in a complex design.

One possible guardian instance for LTTP is discussed in [15, Chapter 5]. There, the main building blocks of a guardian instance are presented from which a particular guardian can be constructed (for both LTTP and FlexRay).

7 Conclusion

The startup problem in fault-tolerant time-triggered communication is often under-estimated. Existing solutions to this problem assume a fail-silent behavior of the nodes in the system. This work, extends the fault-hypothesis to an arbitrary failure model for individual nodes and shows that the startup problem can be solved. This approach can be used to simplify system design and validation of safety-critical systems.

This paper presented a system model for safety-critical systems. Following the system model we discussed a general startup strategy and found that the combination of the assumptions of the system model resulted in the impossibility of steady state detection. This result restricts the design space of a startup algorithm.

From the abstract view on the startup problem we derived a new startup algorithm for LTTP. This algorithm encapsulates fault-tolerance capabilities that increase the freedom of the guardian design. We also analyzed the startup algorithm of FlexRay. This startup algorithm is not inherently fault tolerant and requires high guardian functionality. We used a formal model-checking process, exhaustive fault simulation, to test the startup algorithms in presence of failures. Model checking proofed to be a valuable tool during the construction of fault-tolerant algorithms, which we still use to optimize the startup algorithm and guardian instances.

References

[1] G. Bauer, H. Kopetz, and W. Steiner. Byzantine fault containment in ttp/c. *Proceedings of the 2002 Intl. Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)*, pages 13–16, Jun. 2002.

[2] V. Claesson, H. Lönn, and N. Suri. An efficient tdma start-up and restart synchronization approach for distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.

[3] FlexRay Consortium. *FlexRay Communications System - Protocol Specification - Version 2.0*. FlexRay Consortium, 2004. Available at `http://www.flexray.com`.

[4] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, volume 2788 of *Lecture Notes in Computer Science*, pages 235–248. Springer-Verlag, Vienna, Heidelberg, October 2003.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[6] IEEE, INC. *Standard IEEE 802.4 – Information processing systems– Local Area networks– Part 4: Token-passing bus access method and physical layer specifications*, 1990.

[7] Lindsay Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *IEEE Design & Test of Computers*, pages 4–19, December 1987.

[8] H. Kopetz. *TTP/C Protocol – Version 1.0*. TTTech Computertechnik AG, Vienna, Austria, July 2002. Available at `http://www.ttpforum.org`.

[9] Hermann Kopetz, Michael Paulitsch, Cliff Jones, Marc-Olivier Killijian, Eric Marsden, Nick Moffat, David Powell, Brian Randell, Alexander Romanovsky, and Robert Stroud. Revised version of dsos conceptual model. Project Deliverable for DSoS (Dependable Systems of Systems), Research Report 35/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[10] A. Krüger. *Interface Design for Time-Triggered Real-Time System Architectures*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1997.

[11] H. Lönn. Initial synchronization of TDMA communication in distributed real-time systems. In *19th IEEE Int. Conf. on Distributed Computing Systems*, pages 370–379, Gothenburg, Sweden, 1999.

[12] Henrik Lönn and Paul Pettersson. Formal verification of a TDMA protocol start-up mechanism. In *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*, pages 235–242, Taipei, Taiwan, December 1997. IEEE.

[13] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[14] W. Steiner and M. Paulitsch. The transition from asynchonous to synchronous system operation: An approach for distributed fault-tolerant systems. In *Proceedings of ICDCS*, Vienna, Austria, July 2002. IEEE.

[15] Wilfried Steiner. *Startup and Recovery of Fault-Tolerant Time-Triggered Communication*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2004.

[16] Wilfried Steiner. Model-checking studies of the flexray startup algorithm. Research Report 57/2005, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2005.

[17] Wilfried Steiner, Michael Paulitsch, and Hermann Kopetz. Multiple failure correction in the time-triggered architecture. *Proc. of 9th Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003f)*, Oct. 2003.

[18] Wilfried Steiner, John Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. *The International Conference on Dependable Systems and Networks (DSN 2004)*, Jun. 2004.

[19] C. Temple. Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System. In *Proceedings of 28th Annual International Symposium on Fault-Tolerant Computing*, pages 218–227, June 1998.

[20] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[21] Chris Wellheuser. Metastability performance of clocked fifos: First-in, first-out technology. Technical Report SCZA004A, Advanced System Logic - Semiconductor Group, Texas Instruments, March 1996.