A Fault-Tolerant Dynamic Time-Triggered Protocol

Dissertation
zur Erlangung des Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

von

Dipl.-Inform. Jens Chr. Lisner
aus Mülheim an der Ruhr
lisner@dc.uni-due.de

Vorgelegt beim
Fachbereich Wirtschaftswissenschaften am Campus Essen
der Universität Duisburg-Essen
am 1.10.2007

Kurzfassung

Der zunehmende Einsatz von eingebetteten Netzwerken in sicherheitskritischen Systemen, zum Beispiel in den Bereichen der Automobil- und Luftfahrtindustrie, bringt auch zusätzliche Anforderungen wie Echtzeitfähigkeit und Fehlertoleranz mit sich. In dieser Arbeit wird das neue fehlertolerante Echtzeitprotokoll TEA vorgestellt, das, im Gegensatz zu bereits bestehenden Protokollen, auch die Möglichkeit der fehlertoleranten dynamischen Arbitrierung vorsieht. TEA toleriert Einzel- und Doppelfehler. Um dieses zu realisieren wird ein zeitgesteuertes Verfahren eingesetzt. Dazu wird eine spezielle Hardware-Architektur vorgestellt, in der sich jeweils zwei Netzwerkknoten gegenseitig überwachen. Die Arbitrierung erfolgt in Zyklen, die jeweils in einem regulären und einem erweiterten Teil aufgeteilt sind. Jeder der Controller im Netzwerk kann in dem erweiterten Teil senden, sofern er eine Anfrage im regulären Teil des Zyklus sendet. Ein fehlertolerantes Übereinstimmungsprotokoll entscheidet darüber, ob der Controller letztendlich senden darf oder nicht. Um den Ansatz zu formalisieren, werden zwei neue Methoden vorgestellt, die zur Modellierung der Fehlerausbreitung und des zeitlichen Verhaltens eines Systems dienen. Mit diesen Mitteln wird das Verhalten eines TEA-Netzwerks im Fehlerfall analysiert. Desweiteren werden Wege aufgezeigt, um eine noch dynamischere Nutzung des erweiterten Teils zu ermöglichen, indem die Sendezeit der Controller durch die Nachrichtenlänge bestimmt wird, die dann variabel sein kann. Ausserdem wird ein Schedulingalgorithmus für den erweiterten Teil vorgestellt, der einfach an verschiedene Schedulingstrategien angepasst werden kann.

Abstract

The increasing use of embedded networks in safety-critical systems, for example in the automotive and avionics fields, comes with additional requirements like real-time capabilities and fault-tolerance. This work presents the new fault-tolerant real-time protocol TEA, which provides the ability of fault-tolerant dynamic arbitration, in opposite to already present protocols. TEA tolerates single- and double faults. This is implemented using a time-triggered method. Therefore, a special hardware architecture is presented, where two nodes are guarding each other. The arbitration takes place in cycles, which are separated into a regular and an extension part. Every controller in the network can send in the extension part, if it sends a request in the the regular part of the cycle. A fault-tolerant agreement protocol decides, if the controller gets permission to send or not. In order to formalize this approach, two new methods will be presented, to model the fault propagation and the temporal behavior of a system. Using this methods, the behavior of a TEA-network is analyzed. Furthermore, it will be shown how a more dynamic use of the extension part can be accomplished, by determining the time to send for a controller by the message length which can be variable. Furthermore, a scheduling algorithm for the extension part is presented, which can be easily adapted to different scheduling strategies.

Contents

1 Introduction

1.1 Motivation

In the last two decades, the development in the automotive and avionics fields have evolved an increasing number of electronic components like control devices, sensors an actuators. With the introduction of computer networks in life-critical systems there is a need for highly dependable and fault-tolerant communication in networks with several dozen nodes in modern cars. The widely used CAN networks in automobiles are not sufficient for recent applications. Faults can easily become hazardous in X-by-wire applications, like steer-by-wire or brake-by-wire. Therefore, newer developments with a greater focus on fault-tolerance are gaining increasing success in the industry. Two representatives are the protocols TTP/C and FlexRay. While TTP/C holds a strong focus on fault-tolerance, the developers of FlexRay didn't want to abandon the option to send data without the restrictions which are necessary to protect the network from faults, and therefore preserve higher flexibility during communication. Both protocols use a strict time-triggered pattern to send critical data. This means, the attributes start-time and length of a message are pre-determined. The communication is organized in cycles, which are divided into subsequent slots of equal length (see section 2.4.2). This offers the best conditions for protection of the communication, because collisions can be effectively prevented. However flexibility is poor, because these settings cannot be changed during runtime. FlexRay introduced a mode that allows all nodes an unprotected, but less restrictive communication. In contrast to the static TDMA (time-division multiple access) method, the protocol uses the minislotting scheme in the so-called dynamic segment of the communication cycle. Minislotting is another time-triggered strategy used in protocols in the automotive (e.g. byteflight) and avionics (e.g. ARINC 629) fields (see also 2.4.3).

The FlexRay protocol allows both communication modes in the same application. Therefore, it offers the user a choice: Either sending critical data in a strict pre-determined pattern or using a more flexible, but unprotected way to transmit data.

This work deals with a third option. A new protocol, called TEA, has been developed. It aims to provide flexible communication for critical data as well. While the two transmission modes are independent in FlexRay, TEA uses a static, non-flexible part to prepare the transmission order in the second part of a communication cycle. Once the sequence of senders in the second part is determined, it is also possible to gain flexibility regarding the length of a transmission.

1.2 Problems to be Solved

The TEA protocol focuses on three main topics: Fault-tolerance, flexibility and efficiency.

A protocol provides fault-tolerance, if it assures communication between fault-free components within a network during runtime. This means that a faulty component must not disturb the communication between the fault-free senders and receivers. A possible countermeasure against a faulty communication controller is to avoid access of that controller to the communication media, if it is not allowed to send. Additional devices like guardians, but also other controllers may be used to deny or grant a controller access to the channels. These mechanisms are discussed in full length in chapter 2. A fault-tolerant solution must provide a hardware solution to prevent random access to the channels.

Additional devices which are able to guard a communication controller must be able to determine if a controller is allowed to access a channel,. It is necessary that the sending rights are deterministic. Since static TDMA is fully deterministic (see chapter 2), this approach works best with static TDMA.

Fault-tolerance in static time-triggered protocols like TTP/C is based on a fixed pre-configured behavior of the senders. The drawback of this method lies in the fact that controllers are restricted to the order given by the pre-configured schedule. It is not possible for a controller to send in additional slots within the current communication cycle, if it temporarily needs more bandwidth or redundant communication. On the other side, a controller cannot release bandwidth, if communication is not necessary, because there is no data to be transmitted.

In contrast, there is no similar assumption made in dynamic arbitration methods. It is not known, if a sender is going to send nor how long it will do so. Due to the needs of real-time processing, there are some limitations. To preserve fairness, it is necessary to prevent single senders from monopolizing the communication media. The message size must be limited. The protocol must also avoid collisions. This means, a controller cannot send at any time. Therefore, the term "dynamic" means in turn, that each controller has the right *not* to send, even if it would be possible. In the minislotting protocols ARINC 629 and byteflight, each sender informs the other senders that it will not send immediately, by simply not sending. Therefore it must be clear which sender currently has the right to access the media if it wants to do so. This can be determined either by using a fixed order of possible senders (byteflight) or by a priority-based approach (ARINC 629).

This notion of dynamic arbitration becomes difficult, when fault tolerance properties are required. During the selection process in the current dynamic slot it is necessary for all controllers to agree on the next sender. This is difficult in single- and completely impossible in double-fault scenarios (channel and controller, see section 5.1). Proper communication can only be provided, if any controller can determine, that the sender has accessed the media. If a faulty channel can behave in a byzantine way, one can easily construct scenarios where minislotting fails.

Another important topic is efficiency. Efficiency has two sides in the context of static and dynamic TDMA. In static protocols, it is not possible to use the reserved bandwidth for other purposes, if a controller does not need it. If so, the slot is typically filled with null messages by the owner of the slot. To solve this problem with a TDMA-like protocol it is required to configure additional individual slots for each controller. This enlarges the communication cycle. All changes in the static configuration are permanent throughout the runtime of the network. If a controller needs additional slots rarely, bandwidth is wasted because the sum of the maximum slot numbers for each controller has to be taken into account. Additionally, longer cycles mean greater clock differences near the end of the cycles, since the clocks cannot be synchronized perfectly. So the slots must be chosen longer, to assure that all messages fit into them, and there is enough time to adjust guardians.

On the other side, there is a further kind of overhead in dynamic arbitration with minislotting. Every time the next sender must be selected, the senders may wait for different timeouts (minislots) to expire before they are allowed to send. If this overhead becomes too high, lower prioritized controllers may not be able to send in the same cycle.

It shows up, that the requirements for fault tolerance cannot be fulfilled with the existing minislotting approach, while preserving abilities like fair scheduling and dynamic arbitration. Also, the common minislotting-based protocols cannot grant optimal bandwidth usage.

In this thesis a hardware solution will be provided to allow fault-tolerant communication provided that the schedule of the senders is known. To assure a deterministic schedule also for dynamically allocated slots, TEA provides an agreement algorithm, which is capable of tolerating double-faults. In a network with two channels, one channel and one controller are allowed to be faulty at the same time, without disturbing communication of other controllers in the network.

1.3 Structure of the Thesis

The purpose of TEA is to provide a way to transmit messages over a broadcast network in a fault-tolerant way and providing bandwidth for fault-tolerant communication which can be used on-demand. TEA is designed to tolerate single- and double-faults, which means, that one channel and one controller may be faulty at the same time, but not two controllers or two channels.

Chapter 2 gives an overview about the technical background of the thesis and the current state in development.

This work should also provide theoretical background for presentation and analysis of the required behavior of hardware components in the network. On this basis, the structure of a communication cycle pattern and fault-tolerant message transmission will be established. Chapters 3 and 4 introduce new formal ways to describe fault-propagation in a network of components, and the presentation of the timing of actions as a hierarchy of actions as "timed grammars." The following chapters make use of both methods. While the more hardware and network related chapters 6 and 7 utilize mainly the network behavior functions, timed grammars are used in the protocol related chapters 5 and 9.

Chapter 5 provides an in-detail discussion about the faults to be tolerated and the solution TEA provides. The structure of the communication is defined using the timed grammar formalism from chapter 4.

To achieve fault-tolerant communication TEA lays some constraints on the hardware architecture. The structure of this architecture is the topic of chapter 6. It will be described especially regarding their properties to prevent the propagation of faults using the formalisms developed in chapter 3, and analyzed regarding their ability to prevent the propagation of faults in chapter 7.

The sequence of senders in the flexible part of the communication cycle is determined by an agreement algorithm. It will be described and its correctness be proven in chapter 8.

The remainder of the thesis covers the topics of timing and efficiency, scheduling strategies and strategies for allocation of time-slots with dynamic length. In chapter 9 the timed grammar from section 5 will be used to determine the amount of bandwidth which gets lost due to the overhead necessary to prevent overlapping slots.

The agreement algorithm from chapter 8 is able to select a set of potential senders for the flexible extension part of the communication cycle. The selected senders can send in any order, but in practice a certain strategy for scheduling has to be provided. Chapter 10 shows a low-cost implementation of FIFO- and priority-based or mixed scheduling strategies.

In chapter 11 two different methods, a value- and a timeout-based strategy for providing slots of dynamic length are discussed.

Chapter 12 contains a summary of the thesis, and gives an outlook at future research.

2 Real-Time Communication Protocols

This chapter introduces basic concepts of fault-tolerant real-time communication protocols for distributed systems. Section 2.1 explains the basic idea of real-time distributed systems and the requirements for real-time communication. Section 2.2 introduces some basic concepts of fault-tolerance. The preceding sections 2.3 to 2.6 discuss certain aspects of real-time protocols. This includes a discussion about the advantages and disadvantages of event- and time-triggered strategies and examples of popular arbitration methods. The global time-base, which is one of the driving forces in time-triggered protocols, is the topic of section 2.6. Network architectures are discussed in section 2.5. This covers not only the topology of the network, but also the basic functionality of communication controllers and supporting devices. The chapter concludes with examples for existing protocols and a short discussions about their strengths and weaknesses.

2.1 Real-Time Systems

A real-time computer system is a computer system, whose correctness depends on the correctness of the functional behavior in the value and time domain (for example in [Das85]). The system has to deliver correct results while meeting the timing constraints. Otherwise, a system failure has occurred. Real-time systems are divided into soft real-time systems and hard real-time systems. In soft real-time systems, the timeliness of the results are important, but the expiration of the deadline has no impact on the system safety. On the other side, missing a deadline can have catastrophic consequences in hard real-time systems. Such a system must provide the specified temporal behavior at any stage of operation. According to [Kop97] hard real-time systems are safety-critical systems by definition. A fault-tolerant design is, therefore, important.

A fieldbus network can be one example of a distributed real-time system. The term “fieldbus network” describes a computer network where measurement (sensors) and control devices (actuators) are connected as a local area network (see also [Fou06]). They are used, for example, in automation, automotive and avionics fields. Sensors and actuators are connected to the nodes of the network, which consists (among other things) of a communication controller which implements the protocol.

Protocols like TTP/C, CAN and FlexRay are implementing services on layer 1 (physical layer) and layer 2 (data link layer) of the ISO/OSI reference model ([ISO94], [Zim80]). They specify the underlying hardware constraints, like network topology, node architecture, physical encoding of messages, media access strategy and error detection mechanisms. For real-time systems, it must respect the real-time requirements, like guaranteed maximum latency and deterministic behavior ([CGK+01]). Fault-tolerance is crucial for safety-critical applications. The protocols TTP/C and FlexRay aim to provide communication services even in the case of single-faults. Faults in communication controllers, host controllers and channels should be tolerated.

2.2 Fault-tolerance in Real-Time Communication Protocols

Fault-tolerance is the ability of a system to provide its specified functionality, even if a limited number of components are faulty (see [Ech90]). In [Cri94], two aspects of fault-tolerant systems are identified. A system is said to be fault-tolerant, if its behavior remains well-defined when components fail. On the other side, a fault-tolerant system can mask component failures, so that they are not visible to the user.

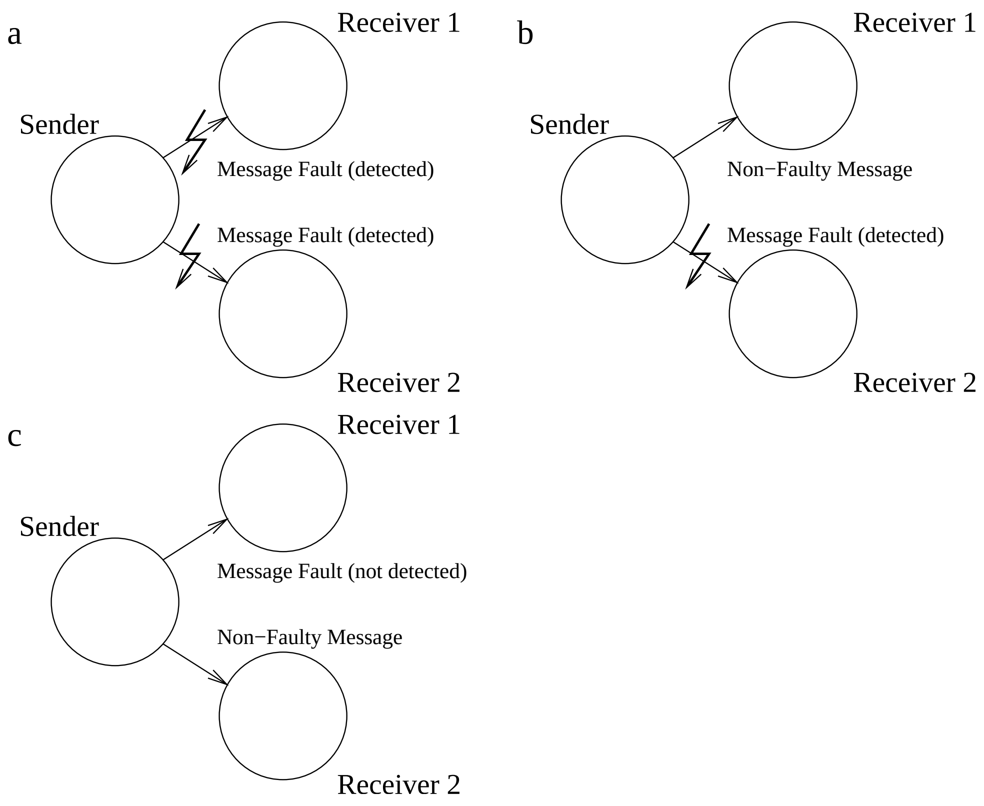


Figure 2.1: In figure a, both controllers can detect a faulty message, whereas one receiver can reliably detect a faulty message in b (benign byzantine case). Figure c shows the malicious byzantine case: Both receivers accept the message from the sender, even if one of the messages is faulty.

A system provides a service to the user or a higher-level system according to its specification. The system has failed, if the service deviates from its specification ([Kop97]). A failure of a system is the consequence of an error state, which is caused by a fault. The relationship between the terms *fault*, *error* and *failure* is explained in [LAK92]. A fault can either be caused by a physical phenomenon, like a broken device or a defective contact, or by a design error, such as a programmer's mistake or an error in the system specification. If a fault affects a system, then the internal state of a system can change in an unintended way. In this case the system enters an error state and the system can fail.

Fault-tolerant systems are designed according to a given fault-model. The fault model describes the structure of a system and the kinds of faults which are possible for the components (see [Ech90]).

The basic dimensions that a fault can affect are value and time (see [Rus01]). A value fault results in an incorrect value. A timing fault causes an event to occur at the wrong time, whether too early, too late or not at all. One way to classify faults is the hybrid fault model described in [TP88]. The effect of a fault can be reliably detected, symmetric meaning that all observers detect the same result, or arbitrary, which means that it is entirely unconstrained. If the effect of the fault is perceived differently by different observers, then the failure is asymmetric or byzantine (see [LSP82]). Figure 2.1 shows a faulty sender and two different receivers and the different situations which can arise.

In a distributed network, the observers can be the nodes receiving a message which has been sent by a sender. Each node could be connected to a channel, for example by a link to one bus (see 2.5.4). If the bus and all links are fault-free, then the message of the sender is delivered correctly to the receivers. Otherwise, if the bus is faulty, then the message may become corrupted, and all receivers detect an error (symmetric case). If, for example, only one link is faulty, then only the node connected by this link may receive a corrupted message, while the others detect no errors (byzantine case).

Byzantine failures can be benign or malicious. A benign byzantine failure is present, if the faulty message can be discovered as faulty. If both receivers obtain different values, but the receiver of a wrong value cannot detect the error, then the failure is malicious.

A component may be active or passive ([Rus01]). While an active component can do anything, a passive component can only change, lose or delay messages. It cannot spontaneously create a new valid message. Checkwords like CRC ([Ham50] and [BHE06]) are frequently used to assure, that a corrupted message becomes always invalid, so that a receiver can detect an error.

2.3 Event- and Time-triggered Media Access

An important aspect for real-time fieldbus protocols is the time when a node is able and allowed to send messages. Different important properties like response time, flexibility, resource utilization, schedulability and predictability, which is very important for fault-tolerance, depend on the media access strategy.

Media arbitration methods can be roughly categorized in event-triggered and time-triggered communication methods. Examples can be found in section 2.4. In event-triggered communication systems, a message is only sent, if there is a demand for it by the host controller of a node. This allows a high degree of flexibility. The media is only allocated as long as needed for the transmission of the message. Depending on the arbitration method, there may be an additional overhead. The CSMA/CD method, for example, requires re-transmission of the message, if there are collisions in the arbitration phase. It is not very likely, but possible, that arbitration fails in such cases. This is not an option for real-time systems, because they require guaranteed time limits. There are other arbitration methods which can determine the amount of time that is needed for arbitration. Aside from this overhead which is required for transmission, maximum utilization of the media is guaranteed.

Predictability is a valuable property for real-time systems. Event-triggered protocols cannot provide temporal predictability on the communication level. For the communication controller it is not possible to guess the occurrence of an event which leads to communication on the channel. It is even not possible to guess how long the media will be allocated. The only way to gain this information is by observing the channel. This also means that a controller is already sending before communication is finally detected. This lack of predictability forbids also to take measures against faulty controllers from accessing the bus, causing collisions and making communication impossible. In addition, the response time is difficult to calculate and can only be constraint by timeouts. Transmission must be held back when other concurrent nodes are accessing the bus first. This introduces an unpredictable overhead in addition to the arbitration overhead. Solutions require scheduling algorithms on application or operating system layers.

If there are applications which require dynamic scheduling (mutual exclusion or precedence for example) the operating system has to decide if the real-time tasks are schedulable and find a feasible schedule. Both operations are NP-hard problems (see [Kop91] and [Mok83]). Therefore, most event-triggered systems use static scheduling in practice.

The limited processing capacity of a communication controller can lead to problems in case of a burst of events (maybe from different nodes). In such scenarios the controller is no longer able to process the continuos stream of messages. This problem can be addressed by some flow control mechanism.

Time-triggered arbitration is done in cycles. Nodes have access to the media in periodic intervals for a certain amount of time. Usually, the time when access is granted is pre-configured. The media is allocated for a node even if there is no data to send. The duration of the period of time for every single controller is greater or equal to the maximum length of a message. So the maximum bandwidth must be reserved for each controller.

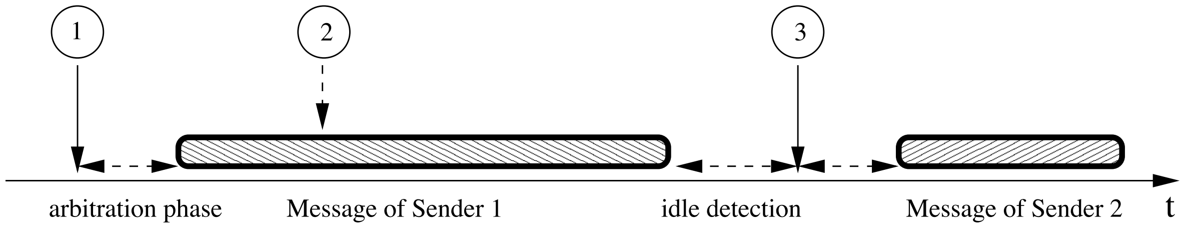


Figure 2.2: If a controller wants to access the media using the CSMA/CA strategy at point ①, the controller has to avoid collisions in the arbitration phase before it can start sending (point ②) in order to avoid collisions. The next sender can try to get sending rights, after the idle state of the media has been detected at ③.

The minimum amount of time between two messages depends on the accuracy of the synchronized clocks. Any controller has to trace the current slot number. To do so the counter of any controller must be increased (or reseted at the end of the cycle) at the same time. This requires a common global time base. Section 2.5.2 shows that there is always a maximum offset δ^c_{max} between the clocks of the nodes. After waiting for the amount of time given by this offset the value of the clock counters of all fault-free clocks are equal. This amount of time can be considered to be the arbitration overhead.

The greatest advantage of the time-triggered approach is its predictability. The time when a sender sends a message is determined at configuration time. This has different consequences. It is guaranteed that each controller can send at least once within a cycle. Dynamic scheduling is not more an issue. Scheduling is a design task in time-triggered systems. Fault-tolerance measures can benefit from predictability. Timing faults can be reliably detected. Knowing what will happen next allows other components in the network to take action against possible faults to occur before the fault can disturb further parts of the system.

As stated out in [Kop93] flow control in time-triggered systems is implicit. The maximum waiting time for a real-time task to send a message can be calculated, even in case of a burst of requests. In case of too few resources (memory capacity), message loss is possible. It is up to the system designer to estimate the requirements of the real-time tasks.

A performance comparison can be found in [CES03]. The average waiting time in time-triggered systems is half a cycle while it depends on load and message sizes in event-triggered systems. Event-triggered systems are best for sporadic message handling with lower loads which implies short message delays and few message losses. The strength of time-triggered systems lies in predictability and dependability. In case of heavy loads, where most time slots are used for message transmission, the system benefits from less overhead.

2.4 Arbitration Strategies

2.4.1 CSMA/CA

The CSMA/CA (carrier sense multiple access / collision avoidance) strategy is used in different protocols ranging from wireless (for example in [Kon06] and [ISO99]) and CAN (see [Bos91]).

In CAN collision avoidance is achieved by address-arbitration. It is assumed, that a recessive and dominant state on a channel exist which are used to code '0' or '1' bits on the channel. A controller can only try to send a message, if there are no communication activities on a channel. In CAN, the controllers are listening 11 bit-times to assure, that no other controller is currently active on the channel (see [Ets01]). Any message has a unique 11- or 29-bit address. The controllers send their address on the channel. If more than one controller is sending, the controller

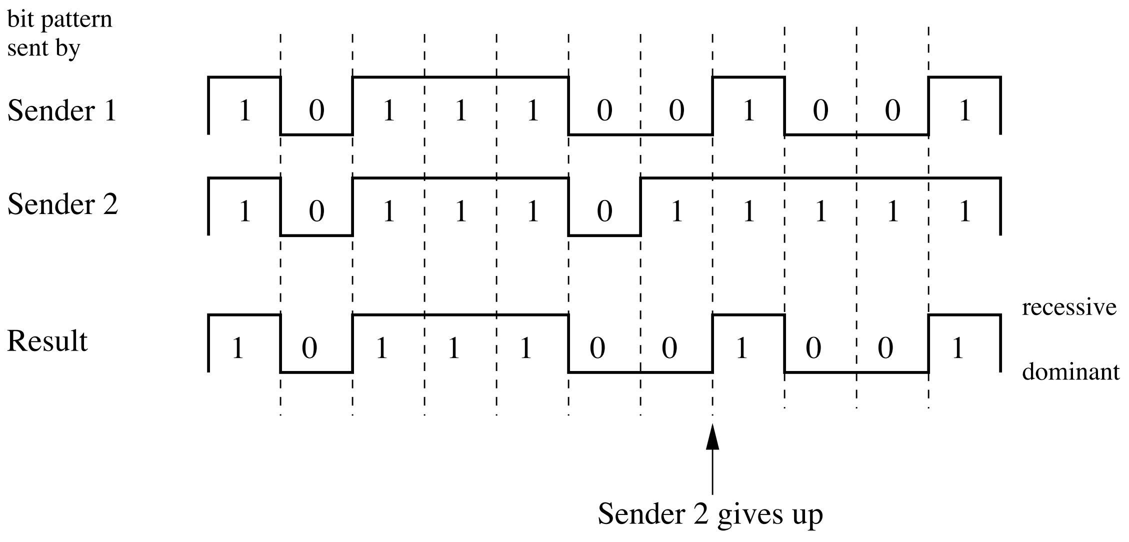


Figure 2.3: The collision avoidance strategy in CAN: The sender with the higher priority (lower identification value) overrides the identification of the sender with lower priority.

with higher priority (which means lower address value), overwrites the address of the other controller. When a sender receives a signal that is different from which it sent, it stops sending, and switches back into “listen only” mode. The winner of the competition gets the right to send on the channel. In the absence of faults collisions are impossible, because the address field is unique for each message.

2.4.2 TDMA

TDMA (time division multiple access) is a time multiplex technique which is widely used in wireless networking, satellite communication and different real-time fieldbus protocols including the time-triggered protocol (TTP – see [KG93] and [KB03]) and FlexRay ([BBE$^+$01]). Since the requirements in wireless and real-time networking are different, the focus on the discussion lies on implementations of fieldbus protocols.

The bus access is arranged in communication cycles, which are subdivided into time-slots of static length. One communication controller is allowed to send in one slot. Each possible sender is assigned to one or more slots. The sender starts sending at the beginning of its slot, and stops sending when the slot ends.

It is common in most protocols to send a fake message (null-frame) which fills the whole slot, if the sender has no data to transmit. These kind of messages are valid messages which contain an empty payload data part. In TTP/C

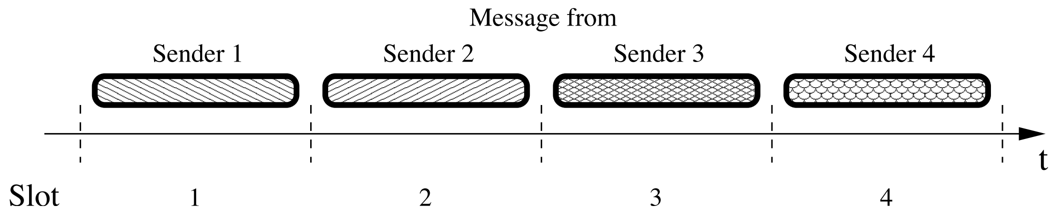


Figure 2.4: A synchronous TDMA scheme using slots with fixed length.

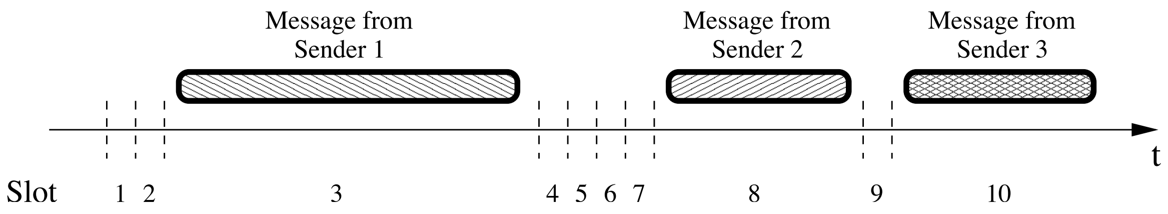


Figure 2.5: In this minislotting scheme each sender is assigned to one slot, but it sends only if there is an demand for it (slots 1, 2, 4 – 7 and 9 are left empty). If a sender starts sending, then the slot is expanded, until the end of the message is reached.

and FlexRay, a null-indicator bit is also set in the header of a null-frame. The reason for using null-frames is that they can also be used for clock synchronization (see 2.6). Additionally, error detection is possible. If a message is missing, then either a fail-omission occurs on the senders side, or the transmission channel is faulty.

The schedule of the senders must be pre-determined before the start of the cycle. Normally this is done at configuration time of the network, and there is no possibility to rearrange the schedule at runtime.

2.4.3 Minislotting

The usage of time slots with static length and fixed schedules in TDMA can become a problem, if throughput, waiting- and response-time become an issue. In TDMA, every controller has to wait a whole cycle, until a new message may be sent, even if some slots are left unused, which means that some controllers send null-frames. One solution to this problem is minislotting. Minislotting allows flexible slot-length, and a significantly shorter waiting time if slots are not used. This is the reason, why it is also called flexible-TDMA (for example [ZG01]).

Minislotting uses communication cycles similar to TDMA cycles. Each controller has one or more unique timeouts, which are multiples of a fixed time value. This time value is called minislot. After sending or receiving a message, each controller which is requested to send by the host controller, sets a timer to one of its timeouts. In fieldbus networks with broadcast channels, the consequence is that each controller sets its timer at the same time to a different timeout. If there is no signal on the channel when the timeout expires, the controller starts sending. In case the controller receives a signal from another controller, the timer is stopped, until the remote controller stops sending. Then, the timer is set anew.

There are various variations on minislotting. Slot counters can be used, which are increased each time a minislot expires. In case that a controller is sending, the slot counter is halted, until the end of the message is detected. In this case, the length of a minislot must be calculated, so that it can be assured that the slot counter has the same value for any controller, when the sender starts sending.

It is also possible to divide the whole cycle into minislots of equal length. In this case, all transmissions are multiples of minislots. In any case, sending is not allowed, if the remaining duration of the cycle is too small to contain the next message.

Minislotting allows more efficient usage of the cycle. The transmissions may have different lengths, and there is no need to send null frames to fill up empty slots. On the other side, controllers with high timeout values may not send, because the time in the cycle that is left at the end is too small to transmit messages. The minislotting scheme gives any controller an implicit priority, since controllers with low timeout values have higher chances to send. In order to avoid that such controllers monopolize the media, some protocols like ARINC 629 ([Moo89] and [AG97]) introduce additional timeouts to deny a controller access to the media for a while after sending.

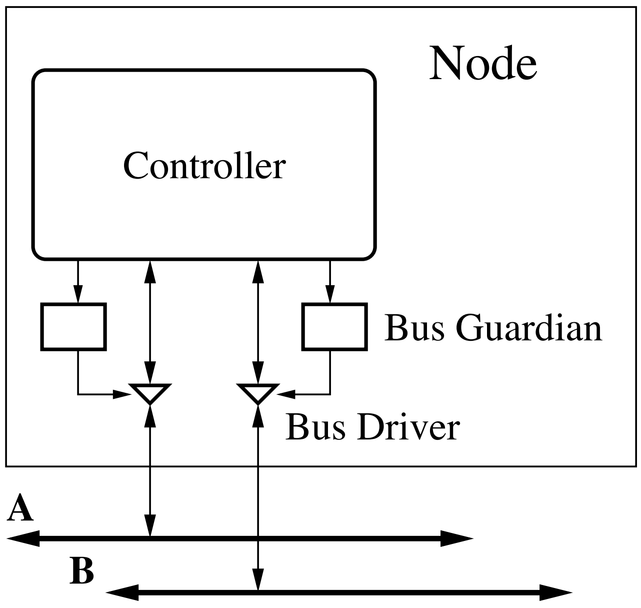


Figure 2.6: An architecture with one controller and two bus guardians as used by the TTP/C and FlexRay protocols.

2.5 Network Architecture

Some protocols go hand-in-hand with a special hardware architecture. An example for this are the guardians in the TTP/C, FlexRay and SAFEbus protocols, which are presented briefly in sections 2.7.4, 2.7.5 and 2.7.3.

Figure 2.5 shows an example for the TTP/C and FlexRay protocols. It shows one node consisting of a controller, two bus guardians to protect the channels in case of faults, and two bus drivers for the connection between the controllers and the channels. In both cases, this is only one variant of possible hardware architectures. Other architectures utilize, for example, central bus guardians, which are possible when using a star topology.

2.5.1 Communication Controller

The core component of a network node is the communication controller.[1] Its responsibility includes encoding and decoding of frames, arbitration, sending and receiving of messages, clock synchronization and error detection and handling.

In the following an overview about the main controller tasks is given. Clock synchronization is discussed in detail in section 2.6. Different arbitration techniques are presented in section 2.4.

Controllers typically provide an interface to the host controller. This includes message buffers for message data to be sent and received, as well as status flags. If the communication controller is allowed to send, then it has to assemble a message and decode it for transmission over the channels. This is done in two steps.

First, a frame is built. Typical frame content is network related information, for example the actual length of the message and the ID of the sender (the message header), the payload data provided by the host in the message buffer

[1]In the following text the terms "controller" is used for "communication controller."

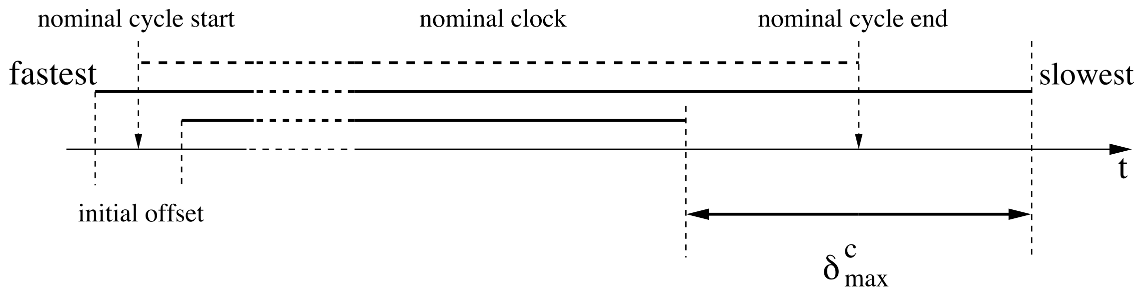


Figure 2.7: The time model is based on the maximum allowed deviation between the clocks of the controllers. Assuming that clock synchronization is done at the end of the cycle, the distance between the slowest and the fastest clock δ^c_{max} is highest.

(the message body) and a checkword, so that the receivers are able to do integrity checks, using techniques like CRC (as in [Fuc95]) or cryptographic signatures.

Then, the binary data must be serialized for transmission over the channels. This physical encoding can be different depending on the physical media which is used for transmission. A two-state media requires a sequence of LOW and HIGH states of a certain length to represent one bit. Sometimes also half-bits are possible. In this case, the bit-time is only half the length of a "regular" bit. Since more than one consecutive bits of the same physical level are possible, and there can be differences in the timing of sender and receiver, some physical encodings require frequent re-synchronization of the receiver to the received message.

It is also often necessary to activate the channel before sending a message, in order to tell all receivers that a message will arrive. This can be done by setting the physical media to a certain level for a certain amount of time.

If the controller acts as receiver, and a transmission is detected, then it has to decode the message. Physical decoding reverses the process of physical encoding. After the bit stream from the channels has been decoded into a frame, the receiver can do integrity checks using the checkwords, and read out the payload into the message buffer. If the checkword test fails, an error state can be signaled.

The question, if a controller can act as a sender or receiver depends on its configuration and the arbitration strategy.

2.5.2 Clocks

Every controller in the network is driven by a clock, whose frequency is derived from a quartz oscillator. This can be a single clock for the whole network (master clock) or a local clock for every controller (distributed clock). The clock can be seen as part of the controller, and may be subject to clock synchronization as described in section 2.6.

The clock maintains a clock time counter, which is increased every time when a certain number of oscillator pulses passed. In the simplest case, the frequency is the same for all nodes. Typically there are differences, depending on quality, age and jitter of the oscillator, so that the real current frequency can only be estimated. Real world protocols specify a maximum deviation from the nominal frequency. It is given for example in parts-per-million (ppm).

The real time (for example measured in ns) at local clock time c of a clock i with a deviation r_i, is given by $T_i(c) = \gamma_i c + \delta^0_{clock,i}$, where $\gamma_i = 1 + r_i$ is the clock rate and $\delta^0_{clock,i}$ the local offset at the beginning of time accounting. The nominal time is represented by a reference clock with $r_i = 0$ and $\delta^0_{clock} = 0$. Figure 2.8 shows the relationship between real-time and local clock time. A negative value for the deviation leads to a faster clock.

The maximum clock deviation value r can be used to identify the slowest clock s and the fastest possible fault-free clock f in the network. The real-time when all fault-free controllers count clock time c lies always between

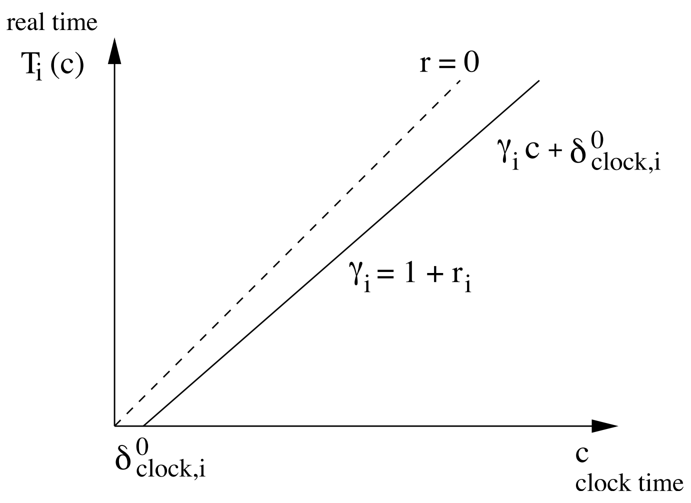


Figure 2.8: The actual speed of a clock i depends on the deviation r_i from the nominal clock speed $r = 0$. The clock starts with an inital offset $\delta^0_{clock,i}$ and has a speed of $\gamma_i = 1 + r_i$. $T(c)$ is the real-time when the local clock reaches time c.

$\gamma_s\, c - \delta^0_{clock}$ and $\gamma_f\, c + \delta^0_{clock}$ where $\gamma_s = 1 + r$ and $\gamma_f = 1 - r$. The maximum offset between two clocks at time c is $\delta^c_{max} = T_s(c) - T_f(c)$ and must always be taken into account, unless all clocks are perfectly synchronized, which is unlikely in real systems.

Since the difference between the local time of the fastest and the slowest clock rises during runtime, the clocks need to be synchronized from time to time. Different strategies for clock synchronization are discussed in section 2.6.

If the clock synchronization works correctly, the highest offset between two clocks is reached just before synchronization. This value is δ^c_{max}.

In practice, there may be a jitter which causes the clock to change its actual speed within a small interval. Taking this jitter into account the actual speed of the clock is $\gamma_i = 1 + r$ with $r \in [r_{min}, r_{max}]$.

2.5.3 Channel Protection

Fault-tolerant architectures need to assure that faulty controllers cannot disturb communications of fault-free controllers. Therefore, the controller can be guarded. A guardian is a device which ensures, that a controller can only access a channel, if it is allowed to do. This section presents different types of guardians, who have all their advantages and disadvantages.

The decentralized guardian concept uses single bus guardians which turn off the bus drivers between the controller and the channels when the controller is not allowed to send. A faulty controller is not able to monopolize the channels and communication remains possible on the channels. In case of a guardian fault, the controller can communicate over the other channel in a dual-channel architecture. Decentralized guardians are located on the same node than the controller. They are typically used in connection with bus- and star-topologies. Decentralized guardians need to know the sequence of senders, and the correct timing. Therefore, a clock must be provided for each bus guardian. Two solutions are possible (see [TTA03], [Tem98] and [Tem99]):

- A fully independent guardian with its own clock synchronization. This is a very expensive approach, because the guardian has to implement its own clock synchronization unit. The preliminary FlexRay bus guardian specification [Fle05a] describes this solution.

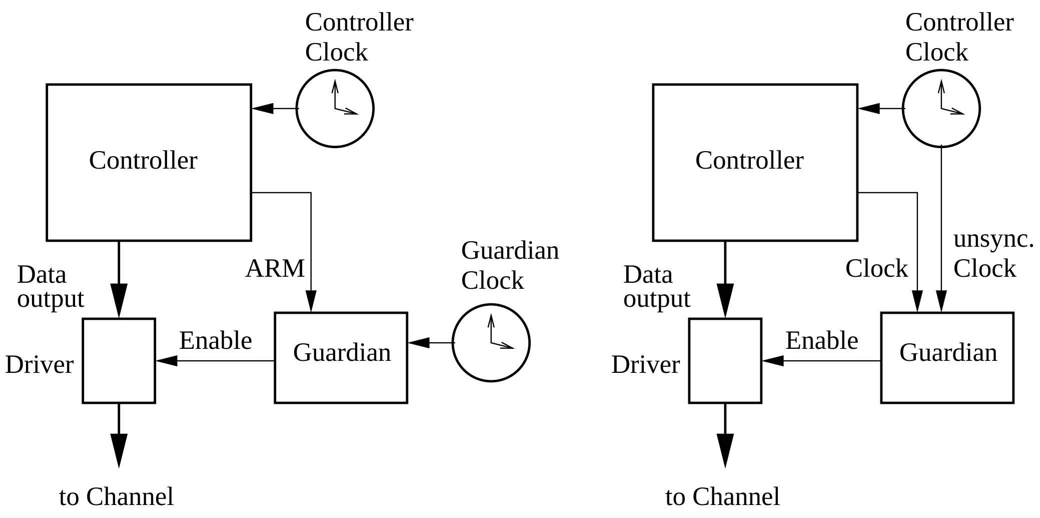


Figure 2.9: Two different bus guardian architectures: The guardian on the left side has an independent clock without clock synchronization. Therefore, it must be re-synchronized in regular intervals by the controller. The guardian on the right side uses two different clock sources: The synchronized clock signal provided by the controller, and the raw unsynchronized signal. Both guardians enable the driver when the controller is allowed to send.

- A guardian with an independent clock source, but without clock syncronization. In this case the clock must be resynchronized periodically with a signal, often referred to as ARM-signal (see figure 2.9), from the controller. Although, a guardian can check if the ARM-signal is observed too far away from the expected time, a misleading controller can cause the guardian to shift the slot boundaries, so that this solution is not fault-tolerant.

Since this is often too expensive, the clocks of the guardians are derived from the same oscillator as the clock of the controller. This introduces a potential single-point-of-failure (see figure 2.9). This variant is proposed in the FlexRay bus guardian specification [Fle04]. Although, this kind of guardian does not use a high-precision clock oscillator, the FlexRay variant provides an optional low-precision oscillator as reference time. In any case, the time base is derived from the controller's clock. The variant shown in figure 2.9 also uses the raw clock signal, to estimate the correctness of the clock synchronization. The bus guardian checks, if the synchronized clock differs too much from the unsynchronized clock.

On the other side, a central bus-guardian consists of a set of bus drivers connected to each controller on one channel. They reside on a star-coupler or hub ([BFJ$^+$00] and [Fle05c]) and can only be used in connection with a star topology. Since one star-coupler is used per channel, the other channel can be used in a dual-channel system in case of guardian faults. A protocol controller maintains a local clock on the central guardian, so that there is a reliable time base. The protocol controller allows access to the channel via the connected bus driver only, if the communication controller requires it.

Another possibility is to group two controllers on one node, and let them guard each other. This is used for example in the SAFEbus architecture (see section 2.7.3). This approach has the advantages, that it can be used with bus- and star-topologies and the timing is very accurate, because both controllers participate in clock synchronization, as the protocol controller in central guardians do, also. The most important disadvantage is, that a fault-free controller can fail to communicate, if its partner controller on the same node is faulty, since a controller controls the connections to all channels of its partner controller. The newer preliminary FlexRay bus guardian specification ([Fle05a]) also mentions the possibility to use a communication controller as bus guardian.

There are other possible solutions based on the guardian concept, especially for ring topologies. In [HDB05]

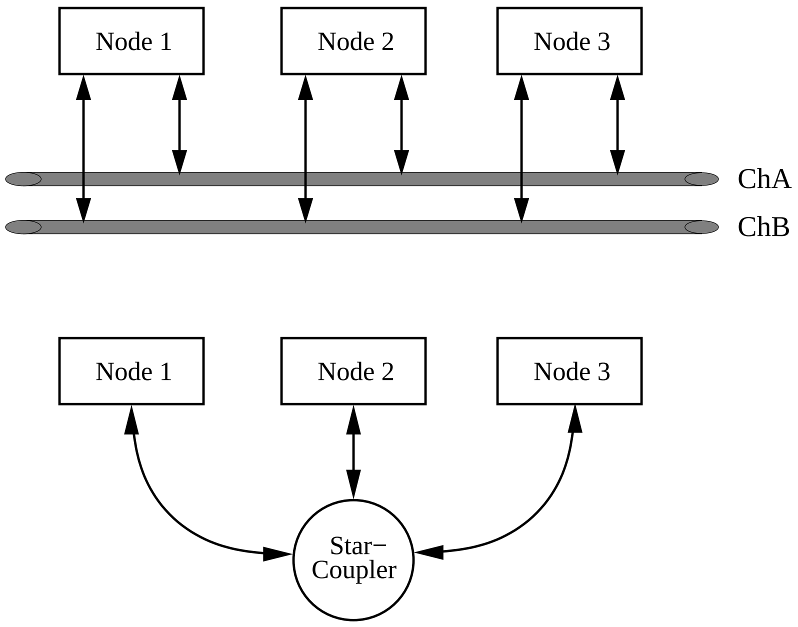


Figure 2.10: Possible broadcast network topologies: A dual channel bus and a single channel star.

an architecture is proposed, where a controller guards the controllers on its respective neighbor node and second neighbor node. This way, one of the receivers is always a fault-free node, which is able to forward a message to the further nodes in the ring.

2.5.4 Network Topology

Fieldbus networks consist of at least one cluster of nodes which are directly connected through channels. Multiple clusters may be connect through gateways. The cluster itself has at least one broadcast communication channel. For redundancy reasons there may be additional channels. The topologies used in both channels are not necessarily the same.

Even though all variants of network topologies could be used in fieldbus networks, bus and star topologies are most frequently used in one cluster (see figure 2.10). Additionally there may be hybrids of both kinds. It is sometimes possible to provide a tree, by cascading multiple stars.

In a star topology the nodes are connected by star couplers. A node is connected to a star coupler by a single link. A star coupler is never used for multiple channels if present because this would undermine the redundancy. While a fault on a single link does no harm to the rest of the network, the channel breaks down if the star coupler is affected as a whole, so it is a single-point-of-failure. Fault-tolerance can then be achieved by providing multiple channels.

A bus topology as in figure 2.10 has no active elements, so collision prevention mechanisms cannot be provided.

Aside from topology considerations, faulty behavior on buses and stars include fail-silent and byzantine behavior. Byzantine behavior occurs for example in case of a network split caused by a faulty link between two star couplers. In any case case, a fault-tolerant network requires at least two distinct channels. Only a central guardian can also actively avoid collisions, if the protocol controller can pre-determine the current sender. Without further mechanisms, this is only possible in time-triggered protocols.

Classic ring topologies are also possible. In a case of a fault-silent node the communication would be impossible. Rings are seldom built in a fault-tolerant way, so they introduce single-point-of-failures. An example for a fault-tolerant ring can be found in [HDB05]. Other examples for ring topologies in real-time networks can be found in [RWS04].

2.6 Clock Synchronization

Exact timing is a crucial problem in distributed systems. A common time base is necessary to provides a unique order of the events in the network protocol. Any oscillator in a node can have small deviations in offset and frequency (see section 2.5.2). Clock synchronization algorithms are necessary to establish a global clock time which is equal for all controllers in the network.

All clock synchronization algorithms have a limited precision. Clock synchronization can only reduce the deviation of the frequency or offsets between the underlying local oscillators. Depending on the quality of the clock synchronization and the maximum deviation of the local clocks, the global maximum deviation of the synchronized time-base is reduced.

2.6.1 Master Clock Synchronization

In master clock synchronization, a single node is responsible for providing timing information to other controllers in the network. The current time may be provided either externally by the host controller, which itself may get the time for example from a time server, or internally in the node by using a high precision oscillator. The current time value can be encoded within a message sent by the controller of the master clock's node, or as a periodic signal. In the latter case all slave nodes adjust their clocks to the edge of the synchronization signal. This can be done, for example, at the beginning of a new cycle in time-triggered systems, or on an extra clock line.

Master clock synchronization introduce a single-point-of-failure. It is possible to detect runaway master nodes, by using the low precision clocks of the slave nodes, but the tolerances of the slaves' clock are higher.

On the other side, this technique can be used in connection with backup masters, which can be activated in the case of a fail-silent master when the synchronization signal is missing.

2.6.2 Distributed Clock Synchronization

It is possible to come around the problem of a single-point-of-failure by using a distributed clock synchronization. This method requires multiple clock masters. Distributed clock synchronization is done in three phases.

In the measurement phase the clock value, which is sent by a clock master, is compared to the value of the own clock. Time-triggered systems with static schedule, for example, can compute the difference between the expected and the arrival time of a message sent by a clock master. It is also possible to send the current clocks' value as part of a message. In any case, the delays, for example for transmission and decoding of the message, must be taken into account. In the case of benign byzantine faults, f faulty senders (clock masters) can be tolerated with $2f + 1$ measurement values. This includes the measurement value of the own clock, which is always set to 0. If malicious byzantine faults are possible, $3f + 1$ measurement values must be available (see [LSP82]).

In the next step, the correction term is computed. This can be done, for example, after each communication cycle or when enough measurement values are available. There are different possible ways how the correction term can be determined. One example is the fault-tolerant midpoint (FTM) algorithm ([Dut98], [LL84]). Based on the number of measurements available, a value k is chosen, so that $2k + 1 \leq m$, where m is the number of measurement values available. Typically, k is limited. In the FlexRay protocol, for example, k cannot become greater than 2 ([Fle05b]).

k is also the number of faults to be tolerated. Then, the k highest and k lowest measured values are discarded. The outcome of the algorithm, the correction value, is the average between the remaining lowest and highest value.

In the following example, the values -12, -7, -4, 0, 5, 8, 10 and 14 should be the measured clock differences (for example the difference between the expected and actual arrival time of a message in clock ticks). With $k = 2$ the list becomes -4, 0, 5 and 8, so that the correction value is $\delta^0_{corr} = \frac{-4+8}{2} = 2$.

Correction can be done by changing the phase (offset correction) or the frequency (rate correction) of a clock. In the first case, the correction term is added to the clock-time counter of the controller. If δ^0_{clock} is the initial offset of the cycle (see section 2.5.2), and the correction is done at the beginning of the cycle, then the controller waits some additional clock ticks, so that the resulting offset becomes $\delta^0_{clock} + \delta^0_{corr}$. Since the correction value can be negative, the cycle schedule should reserve a short amount of time for clock correction. The waiting time is increased or decreased depending on the sign of the correction value.

Phase correction also increases or decreases the length of the cycle in clock ticks. But, the correction value is computed using the difference between two sets of measured values. The reason is, that the correction term δ^g_{corr} is a gradient. In consequence, phase correction can only be done after two consecutive measurement cycles. The number of clock ticks δ^g_{corr} is distributed over the whole cycle, so that the difference between two events within the cycle becomes lower or higher.

So, if $\gamma_c = 1 + r$ is the current clocks' speed (with a deviation r from the nominal clock speed as shown in section 2.5.2), then the clock speed changes to $\gamma_c = 1 + (r + \delta^g_{corr})$ after correction.

A newer method for clock correction is the Daisy-Chain algorithm. The correction takes place after the reception of each frame. This requires a greater safety margin between the frames, but receives better convergence than rate or offset correction. The Daisy-Chain method is presented and analyzed in [Lön99a] and [Lön99b].

2.7 Protocol Examples

2.7.1 CAN

CAN (Controller Area Network) is a fieldbus protocol which was developed 1983 by Bosch for automotive applications. It is one of the most important protocols in automation today. There is also an ISO standard for CAN ([ISO03a], [ISO03b] and [ISO05]).

CAN is an example for an event-triggered protocol. It uses a CSMA/CA arbitration method as described in 2.4.1 (see also [Ets01]). It does not provide any mechanisms for clock synchronization.

CAN provides different error detection mechanisms. Message corruption is detected using 15-bit CRC checkwords. Each controller is able to send a diagnostic error message. Additionally, each controller provides an error counter for self-diagnosis. If the number of detected errors is too high, then a controller can turn itself into *error active* (controller is sending an "active" error flag), *error passive* (controller is sending a "passive" error flag and waits at least 8 bit times before repeating it) and *bus off* (the controller turns off its integrated bus driver).

The protocol cannot assure fault-tolerance in the sense of a binary fault-model, given that the controller is regarded as single fault-region. If the controller behaves arbitrarily, then it can monopolize the channel. CAN nodes have no additional devices such as guardians. Typically, the drivers are integrated within the controller.

The most common topologies used with can are linear buses and stars. In connection with automotive entertainment applications, for example, also ring topologies are supported. In most cases, CAN networks are single-channel networks.

A time-triggered variant of CAN is TT-CAN ([FMD$^+$00], [MFH$^+$02] and [HM$^+$00]), which implements a time-triggered arbitration scheme on top of the CAN protocol. It uses a list of senders to determine the transmission rights. It is also possible to define *arbitration windows*, in which the arbitration follows the regular CSMA/CA scheme. TT-CAN became ISO standard in 2004 ([ISO04]).

2.7.2 byteflight

byteflight is a minislotting based protocol by BMW for automotive applications ([PBG99], [PBG00]). It uses a master-sync clock synchronization, with a high-precision clock. The derivation for the bus master clock should be no more than 100 ppm. byteflight supports star- and bus-topologies.

The protocol provides measures against a fait-silent master clock. In this case, a backup master can be provided. The specification suggests using a parallel network if a higher level of fault-tolerance is necessary ([PBG99]).

2.7.3 ARINC 659/SAFEbus

SAFEbus is the Honeywell implementation of the ARINC 659 standard for usage in commercial airplanes ([Buc03] and [Aer93]). It was also considered to use the protocol for spacecrafts ([GB06]).

The SAFEbus architecture is made highly redundant ([Rus01]). It uses a bus topology with two self checking buses ([HD93]). This means, that each of the two channels consists of two buses that must always show the same state to be considered fault-free. Each bus has a clock line and is two-bit parallel. These are used instead of checkwords to assure message integrity by sending the same bit on both lines, but with reverse signal level on one line. SAFEbus uses a double-controller architecture for channel protection. Each of these controllers, called BUIs (bus interface units), are connected to different lines of the bus-pairs. The nodes are highly complex, and a SAFEbus node is one of the most expensive node which are used for fieldbuses ([Rus01]).

Arbitration in SAFEbus is table driven, which means that the bus access time is divided into windows, and the senders are predetermined before runtime ([HD93]). These windows are of different size from 32 to 8,192 bits. The protocol uses the windows to transmit basic data messages, which are messages containing only data, or master/shadow transfers. Master/shadow transmission is a minislotting technique where nodes with different priorities share the same window.

Clock drift is handled using resync messages (other synchronization message types are used for re-intregation and startup). The sync messages are sent by different senders.

2.7.4 TTP/C

TTP/C is a descendent of the TTP protocol family, originally developed by the Real-Time Systems Group at the Technische Universität Wien ([TTA03]). It uses a strict static TDMA scheme on a dual-channel architecture. The configuration is pre-configured. This assures a high amount of predictability, which can improve fault-tolerance, since all controllers can be sure about the next expected activities in the network. The high degree of fault-tolerance is used to build more reliable implementations of other protocols on top of TTP/C. In [Obe05], for example, the author re-implemented the CAN protocol to gain a higher-level of fault-tolerance for that protocol.

TTP/C requires a double-channel broadcast architecture. Apart from this, the specification imposes only little constraints on the underlying architecture. The TTP/C specification proposes the use of decentralized bus-guardians. The architecture, therefore, allows bus and star-topologies. Buses cannot be used in connection with central guardians.

2.7.5 FlexRay

The FlexRay protocol has been developed by the FlexRay Consortium, which is a group of several companies in the semiconductor and automotive fields.

FlexRay is a time-triggered protocol which operates in communication cycles. In contrast to other protocols, the cycle is split into two segments. In the static segment, the protocol behaves strictly static similar to the TTP/C protocol. The second segment is the dynamic segment, where a minislotting scheme similar to byteflight is used. In this part, the two channels (if available) can be used independently from another.

With FlexRay, a bus-, star- and mixed-topology can be used. Although possible, the FlexRay specification does not demand a double-channel architecture, so that fault-tolerance cannot be assured in case of channel faults when using a single channel. The current bus guardian specification [Fle04] describes decentralized bus guardians sharing the same clock with the controller, but other possibilities are currently under discussion. However, the protocol does not require bus guardians at all. This reduces the hardware costs for a FlexRay node. On the other side, a network without guardians (and maybe only a single channel) does not provide fault-tolerance.

Even if a guardian is present, the protocol requires to open it throughout the whole dynamic segment, so that the channels are completely unprotected against faulty controllers. The idea is, that critical information is sent on the fault-tolerant static segment, while non-critical one can be send in the dynamic segment.

2.8 Summary

The protocols presented in the last sections are examples of different hardware architectures, clock synchronization methods, arbitration methods and fault-tolerance mechanisms. Further examples and detailed comparisons between them can be found in [Rus01], [Kop01] and [GB06].

While CAN represents a typical event-triggered protocol, all other protocols are time-triggered. Both CAN and byteflight do not have a guardian concept or a similar technique to protect the channel from faulty controllers. Also, they are commonly used in single-channel networks, which means that a faulty channel can break down the whole network. Therefore, they don't count as fault-tolerant protocols. On the other side they offer the highest amount of flexibility.

On the other side, TTP/C achieves a high amount of fault-tolerance, because it is designed to be highly predictable. This makes it easy to establish an efficient channel protection mechanism. Due to the dual-channel architecture, it is also able to tolerate single-channel faults. Depending on the guardian architecture chosen, arbitrary node faults can be tolerated as well. The price for this high amount of fault-tolerance TTP/C achieves through its predictability, is the strict pre-defined pattern the whole communication has to follow. So, TTP/C offers the lowest amount of flexibility of all protocol examples presented.

The SAFEbus architecture takes a high effort to achieve fault-tolerance. It utilizes two completely independent controllers (BUIs) for channel protection, which is possible due to the high predictability by using a predefined sequence of message windows. SAFEbus offers also more flexibility at configuration time, by providing windows of different length. Since it is the only protocol that does not use checkwords for message integrity checks, the whole slot can be used for payload data. Hence it is very efficient.

On the other side, the hardware costs are very high. Also, fault-tolerance cannot be achieved with the minislotting-based master/shadow message type, so that most of its flexibility which can be achieved in a fault-tolerant way is limited to configuration time.

FlexRay provides a flexible but non-fault-tolerant dynamic segment, and a possibly fault-tolerant static segment. The amount of possible fault-tolerance depends on the configuration of the hardware architecture. A FlexRay network can be built using a single-channel architecture without any channel protection mechanism. On the other side a high amount of fault-tolerance can be achieved using more expensive configurations. The direction of the currently ongoing discussion about the bus guardian architecture points towards more fault-tolerant solutions like independent or central bus guardians.

The drawback is, that the flexible dynamic segment cannot be used for transmission of critical messages, even in the most redundant architectures. This property is explicitly not aimed at, because the protocol requires the guardians to be open.

It turns out that none of the protocols is able to provide flexibility in a fault-tolerant way. The problem is a trade-off between fault-tolerance, which is achieved in almost all cases by a high amount of predictability and and the flexibility of sending messages on demand, which is per definition non-predictable.

Part I

Methods

3 A Model for Fault-Propagation

This chapter provides a method for modeling and analyzation of fault propagation in a network of components. A faulty component affects the whole system. This means, a faulty component has an impact on other, physically connected, components. Components can handle faults differently. They can be faulty by themselves, or their behavior can be affected by faults propagated by adjacent components.

The modeling technique presented here meets special requirements. These requirements are taken from the later application of the method described in chapter 6.

- "Behavior" can be defined in simple terms:
  It is not necessary to know what is propagated to the adjacent component, but if the propagated information is faulty or fault-free. Value domain issues are solved at a higher level of the TEA protocol.
- Predictable behavior of components:
  The behavior at the output connections of a component should only depend on the behavior at the input connections.
- The model should reflect the physical realities of components and connections. This makes the model more intuitive.
- Time consumption:
  Components need time to propagate messages and signals. In general, time consumption is an important factor in real-time protocols. A model should explicitly state out, that faulty behavior of a component does not have an immediate effect on any other component.

There are different approaches to analyze fault-propagation in a system. A simple solution is to model the fault-free and faulty behavior of components in a system and then tracking the faulty behavior by building up the state space. This can be done using a finite automata. Other methods to analyze the behavior of a system include the use of petri-nets and markovian-chains (for example [But92]) and similar techniques (see also [Rus01]). These techniques are well-known and can be partially automated.

The drawbacks of these techniques are that they focus on systems with complex behavior, which leads to complex models. The focus lies on the behavior of the components itself and the communication between them. The fault-propagation model, which is presented in the following, focuses on faults and the correlation between faults on the incoming and on the outgoing side of a component.

3.1 Component Network

A network of components is given as a *component network* graph. The nodes of the graph represent the components, while the edges stand for the connections between them. The edges are directed, which means that they are either input or output connections. For example, a typical computer architecture consists of components like CPU, bus-controller, memory banks, IO-Controller and mass storage devices. Each component is physically connected to other components.

Definition 1 ***Component network***
A Network graph (network for short) is the directed graph $\mathrm{N} = (\mathcal{K}, \mathcal{V})$*, where* $\mathcal{K} = \{k_1, \ldots, k_n\}$ *is a set of components and* $\mathcal{V} \subseteq \mathcal{K} \times \mathcal{K}$ *a set of connections. Furthermore, the set of input connections* $\mathcal{I}_k \subseteq \mathcal{V}$ *of a component k is defined as the set of all* $(v_i, k) \in \mathcal{V}$*. In a similar way, the set of the output connections* $\mathcal{O}_k \subseteq \mathcal{V}$ *is defined as the set of all* $(k, v_i) \in \mathcal{V}$*.*

In the example above, the interrupt line of the CPU is a single input connection and a part of the input interface of the component representing the CPU in the network graph. Connections where there is incoming and outgoing data transfer like the bus must be represented by two (input and output) edges in the graph.

3.2 Component Compounds

Many systems make use of similar subsystems. A network, for example, may consist of different hosts (servers or terminals) with similar architecture connected by several switches in a star topology. Each star is connected by a gateway to another star. The hosts themselves may also consist of similar subsystems and so on. Network component graphs of such systems have lots of similar repeated patterns within them.

To gain a higher modularity, it is convenient to divide the network graph in subgraphs, which can then be seen as single compounds connected together through their input and output edges (interface).

Definition 2 ***Component compound***
A component compound (compound for short) c is a subgraph $\mathrm{N}_c = (\mathcal{K}_c, \mathcal{V}_c)$ *with* $\mathcal{K}_c \subseteq \mathcal{K}$ *and*

$$\mathcal{V}_c = \left\{(v_i, v_j) \in \mathcal{V} \mid v_i \in \mathcal{K}_c \vee v_j \in \mathcal{K}_c\right\}$$

The input interface is a subset of $\mathcal{V}_c$ *with*

$$\mathcal{V}_{in,c} = \left\{(v_i, v_j) \in \mathcal{V} \mid v_i \notin \mathcal{K}_c \wedge v_j \in \mathcal{K}_c\right\}$$

Similarly, the output interface is also a subset of $\mathcal{V}_c$ *with*

$$\mathcal{V}_{out,c} = \left\{(v_i, v_j) \in \mathcal{V} \mid v_i \in \mathcal{K}_c \wedge v_j \notin \mathcal{K}_c\right\}$$

Note, that a component compound can be part of another compound. Figure 3.1 shows a component network. The dashed boxes denote three possible compounds. Two of them share the same topology and a similar interface.

A compound can be replaced by a new single component, where $\mathcal{I}_k = \mathcal{V}_{c,in}$ and $\mathcal{O}_k = \mathcal{V}_{c,out}$ where k is the new component. This property allows to apply all following definitions not only to single components, but also to component compounds.

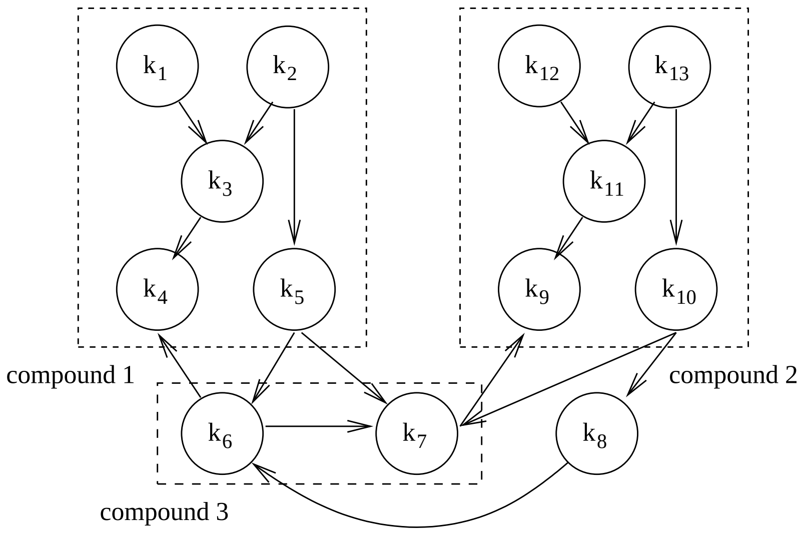


Figure 3.1: A component network with three compounds.

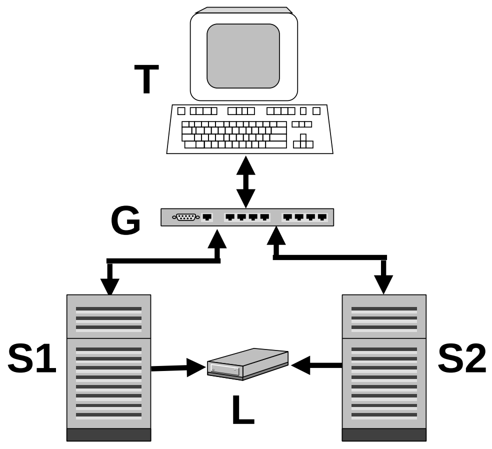


Figure 3.2: A 1-out-of-2 system used in the example 1.

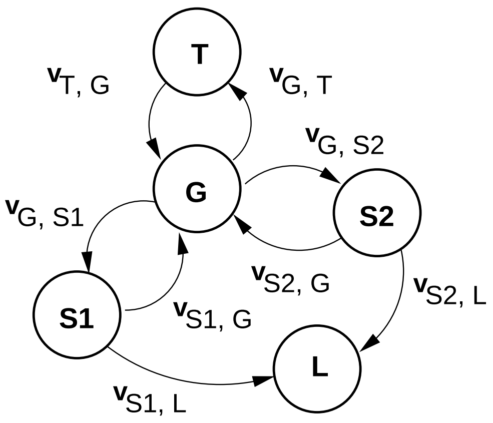


Figure 3.3: A possible network graph for the system in example 1.

3.3 Example 1: A 1-out-of-2 System

Figure 3.2 shows an example system, where a terminal (**T**) can be used to request informations. Two servers (**S1** and **S2**) are processing the request for redundancy reasons. They are connected to a gateway **G** which is able to forward requests from the terminal to both servers and send the reply from the servers back to the terminal. The gateway can check the validity of the answer, and, if necessary, drop a faulty reply from a server while forwarding the fault-free reply (if any). Additionally, both servers are connected to a device **L** which logs every requests at the servers **S1** and **S2**. The servers are responsible to send status messages to **L**.

One possible component network is shown in figure 3.3. The component set is $\mathcal{K} = (\mathbf{T}, \mathbf{G}, \mathbf{S1}, \mathbf{S2}, \mathbf{L})$. All bidirectional connections are splitted into two edges with different directions.

3.4 Protocol and Runtime Activity

During the runtime of a system, a connection in a component network can have one of two states, which are called "active" and "inactive." The activity of a connection is not static and can change during subsequent discrete time steps. The activity of the output connections of a component is called the *component behavior*. It is given by a sequence of states of the network, which is the activity of the connections at time $t \in [t_{start}; t_{stop}]$ and $t_{start}, t_{stop} \in \mathbb{N}_0$ with $0 \leq t_{start} \leq t_{stop} < \infty$.

The *protocol* of a component network shows how a system should behave during runtime between t_{start} and t_{stop}. It can be derived from the system specification. A protocol does not include a data model, except binary signals matching the modeled behavior (active or inactive). Therefore, the intended behavior must be independent from possible information carried with active behavior.

In the example above the specification says, that a request (activity) is followed by a request to both servers which are replying to the gateway. The gateway, then forwards the response (if available) to the terminal. Additionally, a connection to the log device becomes active. At the same time the server sends its response.

Formally, the protocol is a function $\sigma : \mathcal{V} \times \mathbb{N}_0 \mapsto \{0, 1\}$, where $\sigma(v, t) = 1$ denotes activity at connection v at time t. In the other case, the connection v is intended to be inactive.

The protocol of the example system can now be specified as a behavior of the components at their output connections. The table below lists the active edges at a given time beginning with $t_{start} = 1$ and ending with $t_{stop} = 4$.

Time	1	2	3	4
Connections	$v_{\mathbf{T,G}}$	$v_{\mathbf{G,S1}}$ $v_{\mathbf{G,S2}}$	$v_{\mathbf{S1,G}}$, $v_{\mathbf{S1,L}}$ $v_{\mathbf{S2,G}}$, $v_{\mathbf{S2,L}}$	$v_{\mathbf{G,T}}$

This means, when the system starts at time 1 with a terminal request, the connection $v_{\mathbf{T,G}}$ from component **T** to the component **G** is active. It follows, that $\sigma(v_{\mathbf{T,G}}, 1) = 1$ while all other connections at start time are inactive, and so on.

The protocol reflects one certain aspect (in particular the activity of the connections) of the system as specified before run-time. But experience shows, that sometimes the observed behavior of the system does not match the expected behavior as specified. Wrong implementation or hardware faults may cause unexpected results.

Therefore, another function is introduced, to reflect the observed behavior at runtime. This is the runtime activity function $\alpha : \mathcal{V} \times \mathbb{N} \mapsto [0; 1]$, where $\alpha(v, t) = 1$ denotes an active connection $v \in \mathcal{V}$ at time t. Otherwise, if $\alpha(v, t) = 0$, v is inactive at time t. If $0 < \alpha(v, t) < 1$, the activity is “weak.”

It is not always reasonable to assume that activity can be weak. In the example system, the gateway only distinguishes two states. A connection to a server may be active or inactive, but not something in between. But in a different scenario it may be reasonable to have different grades of activity. Assume a battery providing 5 volts for two devices connected in parallel. If the battery becomes faulty, the voltage falls below this value and the devices may stop running. Although the battery does not meet its requirements, one device may also keep running while the other stops depending on the individual tolerances. For example, device 1 may accept 4.5, while device 2 needs at least 4.9 volts. To express this situation in a component network, an individual input threshold is used for the input connections. This topic is addressed later in section 3.10.

With the protocol and runtime activity it is possible to clarify the meaning of expected and unexpected behavior of components. Unexpected behavior can lead to faulty behavior within a system.

3.5 Expected and Unexpected Behavior of Components

When a component shows the same runtime behavior on its outputs as intended, then the component is said to behave as expected. Otherwise, the component behaves unexpectedly at a given time on a given output edge. Only a faulty component can behave unexpectedly. Note, that this statement is not necessarily true the other way around. A faulty component may behave as expected from a fault-free component. On the other side, expected behavior means always fault-free behavior. The same is true for unexpected behavior which is always faulty behavior.

The component behavior is a function, which assigns each outgoing connection a value between 1 and 0, depending on the kind of results on this connection. A value lower than 1 means unexpected behavior.

Definition 3 ***Component behavior***
The component behavior of a component $k \in \mathcal{K}$ is given by the function $\kappa : O_k \times \mathbb{N} \mapsto [0; 1]$.

A component behaves as expected at output edge $v_i \in O_k$ at time $t \in [t_{start}; t_{stop}]$, if $\kappa(v_i, t) = 1$. Otherwise, it behaves unexpected.

As said above, a component should be said to behave as expected if the intended behavior, as defined by the protocol, is the same as the runtime behavior. This means, that $\kappa(v_i, t) = 1 \Leftrightarrow \sigma(v_i, t) = \alpha(v_i, t)$. Therefore, the component behavior can be defined as

$$\kappa(v_i, t) = \alpha(v_i, t)\,\sigma(v_i, t) \;+\; (1 - \alpha(v_i, t))\,(1 - \sigma(v_i, t))$$

The set $\mathcal{F}_{k,t}$ characterizes the faulty behavior of a component. It is the set of all outputs of a component, that show unexpected behavior. In case of a faulty component, the set O_k can be partitioned into two sets $\mathcal{F}_{k,t} \subseteq O_k$ and $\overline{\mathcal{F}_{k,t}} = O_k - \mathcal{F}_{k,t}$ at a given time t, where

$$\mathcal{F}_{k,t} = \{v_i \in O_k \mid \kappa(v_i, t) < 1\}$$

Faulty behavior of a component can have more or less impact on all of its outputs. This can be measured by the number of outputs showing faulty behavior per total number of outputs, which is

$$\Upsilon_{k,t} = \frac{|\mathcal{F}_{k,t}|}{|O_k|}$$

Three different cases of component behavior can be identified:

- If $\Upsilon_{k,t} = 0$, the component behaves correctly at time t. The component may be fault-free or faulty.
- If $0 < \Upsilon_{k,t} < 1$, the component behaves unexpectedly on some of the outgoing connections, but behaves expectedly on others.
- The component k behaves unexpectedly on all outgoing connections at time t, if and only if $\Upsilon_{k,t} = 1$.

Depending on the topic of interest, there are types of faults which are not regarded as critical. This may be true, for example, in the case that a connection is currently not active, although it should be. If inactivity is supposed to be non-destructive and the component behavior function κ can be replaced by the function

$$\kappa'(v,t) = \kappa(v,t) + (1 - \alpha(v,t))\,\sigma(v,t)$$

Similarly, if a connection is allowed to be active even if it should not be, the component behavior is

$$\kappa''(v,t) = \kappa(v,t) + (1 - \sigma(v,t))\,\alpha(v,t)$$

If the activity is weak, then the component behaves faulty. Higher values mean that the differences between the activity value and the value specified in the protocol are lower. The following table shows examples with the values 0.3 and 0.7 for $\alpha(v_i, t)$.

$\alpha(v_i,t)$	$\sigma(v_i,t)$	$\kappa(v_i,t)$	$\kappa'(v_i,t)$	$\kappa''(v_i,t)$
0.3	0	0.7	0.7	1
0.3	1	0.3	1	0.3
0.7	0	0.3	0.3	1
0.7	1	0.7	1	0.7

3.6 Network Behavior

The final behavior of a component in the network does not only depend on the state of the component, but also on the behavior of the adjacent components at the incoming connections. The purpose of a relay, for example, is to close a circuit, if the coil is energized. A non-faulty relay will show this behavior, because it is the behavior which is expected. But from a system point of view the behavior may be wrong, if the adjacent components connected with the inputs of the relay show unexpected (or faulty) behavior. This means, that a fault-free (non fault-tolerant) component propagates faulty behavior from its input connections to one or more output connection. The *network behavior* is a function, which takes both parts into account. It characterizes the operation of a component at a given time depending on the component itself and its inputs.

Definition 4 ***Network behavior***
If $v \in \mathcal{V}$ is a connection in a component network graph, then there is a network behavior function $\phi : \mathcal{V} \times \mathbb{N} \mapsto [0; 1]$, where $\phi(v, t)$ gives the error state of v at time t.

If the component behavior $\kappa(v, t_1)$ is a factor or summand in $\phi(v, t)$, then $t_1 = t$. If $v \in O_k$ and $\phi(w, t_2)$ is a factor or summand in $\phi(v, t)$, then $w \in \mathcal{I}_k$ and $t_2 \leq t$. In this case, v depends on w.

Network behavior functions can be different for different connections of the same component. The shape of the functions of all output connections of a component characterizes the component and the internal relation between its inputs and outputs. In the simplest case, the set of input connections $\mathcal{I}_k$ for component k is empty. In this case the network behavior can be the same than the component behavior, so that $\phi(v_i, t) = \kappa(v_i, t)$, or the network behavior can be a constant expression (either 0 or 1) for $v_i \in O_k$. This function is also true, if v_i is independent from any $v_j \in \mathcal{I}_k \neq \emptyset$.

A simple example for this is a network hub, which has a monitoring output (power on – power off) and some inputs for network cables. It should be clear, that these inputs don't have any impact on this output, so that the network behavior for this output is independent from the inputs in the component network. On the other side, the outputs to the network depend on the correct behavior at the inputs.

Output connections (possibly of the same component) may depend on some or all of its inputs. In the last case (an output connection depends on all input connections), the network behavior of a connection $v_i \in O_k$ may be

$$\phi(v_i, t) = \kappa(v_i, t) \prod_{j=0}^{|\mathcal{I}_k|} \phi(v_j, t)$$

Fault-tolerant behavior can also be expressed as network behavior. If v_i is an output, and v_j an input connection, a component can show fault-tolerant behavior at v_i when $\phi(v_i, t) = \kappa(v_i, t)\ (\phi(v_j, t) + (1 - \phi(v_j, t)) = \kappa(v_i, t)$. Faulty behavior at input v_j does not have an impact on the output connection v_i. On the other side, the component can show faulty behavior at v_i, if it is faulty itself. This is the reason for the $\kappa(v_i, t)$ which remains as the network behavior $\phi(v_i, t)$.

Most, except for the simplest, components have a delay. This means, that changes at the input connections do not affect the network behavior immediately. A simple example for a delay is the signal transfer delay over an electrical media. The network behavior function must take this into account. A possible function could be $\phi(o, t+d) = \kappa(o, t+d)\,\phi(i, t)$, where o is the outgoing, i the incoming connection and $d > 0$ the delay. Faults caused by the component itself affect the outgoing connection immediately. For this reason, $t+d$ is used in the component behavior part of the function.

3.7 Necessary Constraints on Network Behavior Functions

All network behavior functions should provide a well defined value at any time. This means, that ...

1. there must be a well defined initial state at all connections at t_{start}, and
2. if there are cyclic dependancies between the network behavior at different connections, the time at the beginning of the cycle must be different from the time at the end of the cycle.

The first point is obvious. The second point should be illustrated by the following scenario.

Figure 3.4 shows a simple component network with three components k_1, k_2, k_3 and k_4, and four connections v_1, v_2, v_3 and v_4. The network behavior for each connection is

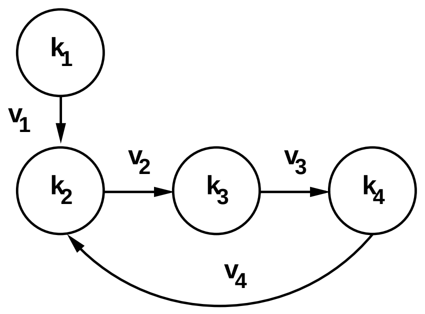


Figure 3.4: A component networks with a cycle.

v_1	$\phi(v_1, t) = \kappa(v_1, t)$
v_2	$\phi(v_2, t) = \kappa(v_2, t)\ \phi(v_4, t)\ \phi(v_1, t)$
v_3	$\phi(v_3, t) = \kappa(v_3, t)\ \phi(v_2, t)$
v_4	$\phi(v_4, t) = \kappa(v_4, t)\ \phi(v_3, t)$

It is not possible to advance from t_{start} to t_{stop}, if $t_{start} \neq t_{stop}$. It is even not possible to start the recursion, because the network behavior at any time including t_{start} cannot be determined.

On the other side, the network behavior can be determined at any connection at any time, if there is a delay in one connection in the cycle, and an initial value at t_{start} is given for that connection. The following network behavior function shows an example for connection v_2:

$$\phi(v_2, t) = \begin{cases} \kappa(v_2, t)\ \phi(v_1, t) & \text{for } t = 1 \\ \kappa(v_2, t)\ \phi(v_4, t-1)\ \phi(v_1, t-1) & \text{otherwise} \end{cases}$$

The network behavior at all connections is well defined, because it is independent from $\phi(v_4, t_{start})$, so that the cyclic dependancy at t_{start} is broken. The result for t_{start} is, therefore:

$$\begin{aligned} \phi(v_1, t_{start}) &= \kappa(v_1, t_{start}) \\ \phi(v_2, t_{start}) &= \kappa(v_2, t_{start})\ \kappa(v_1, t_{start}) \\ \phi(v_3, t_{start}) &= \kappa(v_3, t_{start})\ \kappa(v_2, t_{start})\ \kappa(v_1, t_{start}) \\ \phi(v_4, t_{start}) &= \kappa(v_4, t_{start})\ \kappa(v_3, t_{start})\ \kappa(v_2, t_{start})\ \kappa(v_1, t_{start}) \end{aligned}$$

The result for $t + 1$ is

$$\begin{aligned} \phi(v_1, t+1) &= \kappa(v_1, t+1) \\ \phi(v_2, t+1) &= \kappa(v_2, t+1)\ \kappa(v_1, t+1)\ \phi(v_4, t) \\ \phi(v_3, t+1) &= \kappa(v_3, t+1)\ \kappa(v_2, t+1)\ \kappa(v_1, t+1)\ \phi(v_4, t) \\ \phi(v_4, t+1) &= \kappa(v_4, t+1)\ \kappa(v_3, t+1)\ \kappa(v_2, t+1)\ \kappa(v_1, t+1)\ \phi(v_4, t) \end{aligned}$$

This means, that cycles must include a delay and all network behavior functions with a delay must provide an initial state. This also assures that a system can stop at t_{stop} which ends the recursion.

In the following, if there is no initial state explicitly given for a network behavior function with a delay d, then it should be implicitly $\kappa(v, t)$ for a connection v and $t \in [t_{start}; t_{start} + d - 1]$.

3.8 Results

Network behaviour functions can express the behavior (fault-free or faulty) of a component at its output connections. On the other side, a component without outgoing connections has no behavior. As a substitute, it is considered, that such a component has a “final state” depending on the behaviour at the input connections and one or more result functions.

Result functions have similar properties as network behaviour functions, except that they are not defined for a particular connection. There may be an arbitrary number of result functions for a component. If a component has two input connections, and the result should be fault-free if both connections show fault-free behavior, then the following behavior function can be used, where k is the component.

$$\mathrm{RES}(k, t) = \phi(v_1, t)\,\phi(v_2, t)$$

with $v_1, v_2 \in \mathcal{I}_k$. Note, that the component behaviour is not part of the result, because a component with no outgoing connections doesn't have a behavior. This is another difference between result functions and network behavior functions.

Result functions can also be defined for regular components (components with behaviour). This gives the opportunity to monitor the behavior at a component independently from the values propagated to the output connections.

3.9 Example 2: Network Behaviour of a 1-out-of-2 System

In this section, the network behaviour will be developed for the example system in section 3.3, starting with the terminal device **T**. It has one output connection to the gateway, and the request it sends does not stand in relationship to its input connection. Therefore, the network behaviour of the terminal at this output connection is the component behavior of the terminal, so that $\phi(v_{\mathbf{T},\mathbf{G}}, t) = \kappa(v_{\mathbf{T},\mathbf{G}}, t)$.

Both, server one and two share the same properties. The two output connections depend both on the input connection. Additionally, it is assumed, that both servers need the same amount of time to produce a result. This delay is set to one. The following functions for **S1** are also used for **S2**.

$$\phi(v_{\mathbf{S1},\mathbf{L}}, t+1) = \kappa(v_{\mathbf{S1},\mathbf{L}}, t+1)\,\phi(v_{\mathbf{G},\mathbf{S1}}, t)$$

$$\phi(v_{\mathbf{S1},\mathbf{G}}, t+1) = \kappa(v_{\mathbf{S1},\mathbf{G}}, t+1)\,\phi(v_{\mathbf{G},\mathbf{S1}}, t)$$

The gateway **G** has three output connections. Both, the connection to **S1** and **S2** depend on the input from the terminal. It is assumed that a gateway forwarding causes a delay by one, so that

$$\phi(v_{\mathbf{G},\mathbf{S1}}, t+1) = \kappa(v_{\mathbf{G},\mathbf{S1}}, t+1)\,\phi(v_{\mathbf{T},\mathbf{G}}, t)$$

and

$$\phi(v_{\mathbf{G},\mathbf{S2}}, t+1) = \kappa(v_{\mathbf{G},\mathbf{S2}}, t+1)\,\phi(v_{\mathbf{T},\mathbf{G}}, t)$$

The gateway is also a component which shows fault-tolerant behaviour. It can tolerate fault-free network behaviour on both inputs from **S1** and **S2** (which can be expressed by $\phi(v_{\mathbf{S1},\mathbf{G}}, t)\,\phi(v_{\mathbf{S2},\mathbf{G}}, t)$), faulty network behaviour on the connection from server **S1** (expressed by $(1-\phi(v_{\mathbf{S1},\mathbf{G}}, t))\,\phi(v_{\mathbf{S2},\mathbf{G}}, t)$) and faulty network behaviour on the connection from the server **S2** (expressed by $\phi(v_{\mathbf{S1},\mathbf{G}}, t)\,(1-\phi(v_{\mathbf{S2},\mathbf{G}}, t))$). Faulty behaviour on both input connections cannot be tolerated.

The final function for the connection from **G** to **T** is:

$$\phi(v_{\mathbf{G},\mathbf{T}}, t+1) = \kappa(v_{\mathbf{G},\mathbf{T}}, t+1)\,(\phi(v_{\mathbf{S1},\mathbf{G}}, t)\,\phi(v_{\mathbf{S2},\mathbf{G}}, t) + (1-\phi(v_{\mathbf{S1},\mathbf{G}}, t))\,\phi(v_{\mathbf{S2},\mathbf{G}}, t) + \phi(v_{\mathbf{S1},\mathbf{G}}, t)\,(1-\phi(v_{\mathbf{S2},\mathbf{G}}, t)))$$

Additionally, there are three results. The results for logging the servers **S1** and **S2** are independent from each other. This means, that there are two different results for the log device **L**. These are

$$\mathrm{RES}_1(\mathbf{L}, t) = \phi(v_{\mathbf{S1},\mathbf{L}}, t)$$

and similarly

$$\mathrm{RES}_2(\mathbf{L}, t) = \phi(v_{\mathbf{S2},\mathbf{L}}, t)$$

It is expected that the process ends at the terminal at time t_{stop}. The result function is $\mathrm{RES}(\mathbf{T}, t) = \phi(v_{\mathbf{G},\mathbf{T}}, t)$.

If the result at $\mathrm{RES}(\mathbf{T}, t_{stop}) = 1$, then the result of the process is fault-free. This means, that the user at the terminal received an answer from at least one server.

3.10 Input Threshold of Connections and Acceptance

If a component shows weak activity on a connection, then the target component may regard the behaviour as fault-free. This is often the case, for example, if fault-free components show timing or measurement tolerances at their inputs. For each connection v the *input threshold* consists of two values $a_0(v)$ and $a_1(v)$ between 0 and 1. Both values determine the input threshold depending on the protocol of the system. The *acceptance function* determines, if a component tolerates faulty behavior at input connection v at time t.

Definition 5 ***Acceptance function***
The acceptance function, is a function $\theta : \mathcal{V} \times \mathbb{N} \times \mathbb{R} \times \mathbb{R} \mapsto \{0, 1\}$, *which is defined as follows:*

$$\theta(v, t, a_0(v), a_1(v)) = \begin{cases} 1 & \text{, if } (\sigma(v,t) = 0 \land \phi(v,t) \geq a_0(v)) \lor (\sigma(v,t) = 1 \land \phi(v,t) \geq a_1(v)) \\ 0 & \text{, otherwise} \end{cases}$$

The function shows, that the activity at the input v is accepted for values near the expected activity. In case of expected inactivity, the network behavior must be at least $a_0(v)$, or $a_1(v)$ in the active case.

In a binary model, where the components allow no tolerances (like the example in section 3.3), all $a_0(v) = a_1(v) = 1$ for all connections v at any time. The following section shows an example with different input thresholds.

An acceptance function can be used anywhere, where a network behavior function can be used also. It can therefore be part of a network behavior function or a result function.

3.11 Example 3: Input Threshold

Figure 3.5 shows the component network of a simple system with one sender and two receivers connected to the same component. **S** should send a signal to **F**. This signal is forwarded to receivers **R1** and **R2** with no delay. The following table summarizes the protocol:

Time	0–2	3–6	7–11	12–15	16–20	21–24	25
$v_{\mathbf{S},\mathbf{F}}$	0	1	0	1	0	1	0
$v_{\mathbf{F},\mathbf{R1}}$	0	1	0	1	0	1	0
$v_{\mathbf{F},\mathbf{R2}}$	0	1	0	1	0	1	0

To decode the message, both receivers have to measure the signal on the bus. In a realistic scenario, both receivers have different properties in determining the level of the signal. Figure 3.6 shows an example. Signal level HIGH

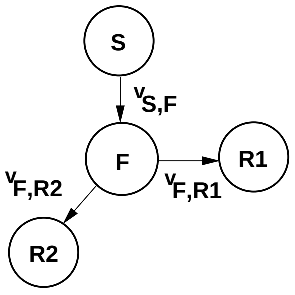


Figure 3.5: A component network with sender and receivers.

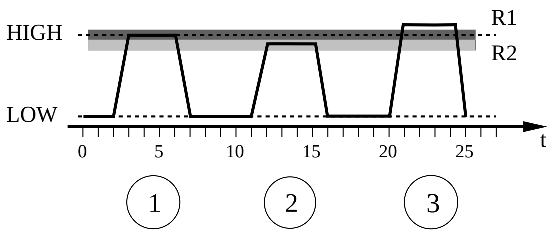


Figure 3.6: A signal is measured by two receivers with different inaccuracies.

means activity, LOW means inactivity. There are three intervals where the connections must show activity (marked as ①, ② and ③).

In this example, the sender **S** is considered to be faulty.

The network behavior and result functions should be:

$$\begin{aligned}
\phi(v_{\mathbf{S},\mathbf{F}}, t) &= \kappa(v_{\mathbf{S},\mathbf{F}}, t)\\
\phi(v_{\mathbf{F},\mathbf{R1}}, t) &= \kappa(v_{\mathbf{F},\mathbf{R1}}, t)\ \phi(v_{\mathbf{S},\mathbf{F}}, t)\\
\phi(v_{\mathbf{F},\mathbf{R2}}, t) &= \kappa(v_{\mathbf{F},\mathbf{R2}}, t)\ \phi(v_{\mathbf{S},\mathbf{F}}, t)\\
\mathrm{RES}(\mathbf{R1}, t) &= \phi(v_{\mathbf{F},\mathbf{R1}}, t)\\
\mathrm{RES}(\mathbf{R2}, t) &= \phi(v_{\mathbf{F},\mathbf{R2}}, t)
\end{aligned}$$

Obviously, the result is the same for both components **R1** and **R2**, because $\kappa(v_{\mathbf{F},\mathbf{R1}}, t) = \kappa(v_{\mathbf{F},\mathbf{R2}}, t) = 1$ since **F** is fault-free.

$$\begin{aligned}
\mathrm{RES}(\mathbf{R1}, t) &= \kappa(v_{\mathbf{F},\mathbf{R1}}, t)\ \kappa(v_{\mathbf{S},\mathbf{F}}, t)\\
&= \kappa(v_{\mathbf{S},\mathbf{F}}, t)\\
\mathrm{RES}(\mathbf{R1}, t) &= \kappa(v_{\mathbf{F},\mathbf{R2}}, t)\ \kappa(v_{\mathbf{S},\mathbf{F}}, t)\\
&= \kappa(v_{\mathbf{S},\mathbf{F}}, t)
\end{aligned}$$

This means, the results for both receivers depend on the runtime activity at $v_{\mathbf{S},\mathbf{F}}$. Figure 3.6 shows that the activity does not match the protocol requirements at ② and ③. In particular $0 < \alpha(v_{\mathbf{S},\mathbf{F}}, t) < 1$ for $t \in [12; 15]$ or $t \in [21; 24]$. Note, that the runtime activity is lower than 1 in ③ say $\alpha(v_{\mathbf{S},\mathbf{F}}, t) = 0.8$ (see also table below), although the signal level is higher than HIGH, which has been identified as activity which has the value 1. Runtime activity is always lower than one if the signal does not match the correct HIGH level.

In this example $\alpha(v_{\mathbf{S},\mathbf{F}}, t) = 0.9$ at ② and 0.8 at ③. Without considering an acceptance thresholds, both receivers detect faulty behavior at ② and ③.

The situation changes regarding the following situation: Both **R1** and **R2** measure a signal to determine its validity. The process of measurement is done with an inaccuracy in this example $a_1(v_{\mathbf{F},\mathbf{R1}}) = 0.95$ and $a_1(v_{\mathbf{F},\mathbf{R2}}) = 0.85$. The following tables give an overview about the current situation:

Interval	$\sigma(v_{\mathbf{S},\mathbf{F}}, t)$	$\alpha(v_{\mathbf{S},\mathbf{F}}, t)$	$\kappa(v_{\mathbf{S},\mathbf{F}}, t)$	$a_1(v_{\mathbf{F},\mathbf{R1}})$	$a_1(v_{\mathbf{F},\mathbf{R2}})$
①	1	1	1	0.95	0.85
②	1	0.9	0.9	0.95	0.85
③	1	0.8	0.8	0.95	0.85

The result functions should be changed to:

$$\begin{aligned}\mathrm{RES}(\mathbf{R1}, t) &= \theta(v_{\mathbf{F},\mathbf{R1}}, t, 1, 0.95)\\ \mathrm{RES}(\mathbf{R2}, t) &= \theta(v_{\mathbf{F},\mathbf{R2}}, t, 1, 0.85)\end{aligned}$$

with $a_0(v_{\mathbf{F},\mathbf{R1}}) = a_0(v_{\mathbf{F},\mathbf{R2}}) = 1$.

Figure 3.6 shows two gray areas around the HIGH level, which show the inaccuracy of the measurement for the receivers. At ① the signal is accepted by both receivers, because **S** shows fault-free behavior, and $a_1(v_{\mathbf{F},x}) < 1$ for both receivers $x \in \{\mathbf{R1}, \mathbf{R2}\}$. On the other side, $a_1(v_{\mathbf{F},x}) > \kappa(v_{\mathbf{S},\mathbf{F}}, t) = 0.8$ at ③, so that the signal is rejected by both **R1** and **R2**. But at ②, the situation for **R1** and **R2** is different. **R1** rejects the signal, because $a_1(v_{\mathbf{F},\mathbf{R1}}) > \kappa(v_{\mathbf{S},\mathbf{F}}, t) = 0.9$. But the signal is accepted by **R2**, which has a higher tolerance $a_1(v_{\mathbf{F},\mathbf{R2}}) = 0.85 < 0.9$. This situation results in a byzantine fault.

3.12 Automated Evaluation

Modeling fault-propagation with network behavior functions has the advantage, that different fault-scenarios can be analyzed automatically by software tools. To investigate concrete scenarios with given values, it is necessary to provide a value for $\kappa(v, t)$ for all $v \in \mathcal{I}_k$ and all $t \in [t_{start}; t_{stop}]$ if k is a faulty component. Then, each network behavior value can be determined by traversing the network component graph recursively with $\kappa(v_l, t) = 1$ for all fault-free component l at any time.

More general investigations are possible. A binary analysis does not work with certain given values. This introduces a the problem, that it is not possible to determine the outcome of an acceptance function if the behavior at the respective input connection is faulty. Therefore, multiple solutions are possible for the same timestep.

Investigations of that kind are relatively efficient depending on the network. If there are $n = |\mathcal{K}|$ components, then the maximum number of connections is n^2. Since the maximum time that is needed to evaluate one component behavior function is constant, the complexity is $O(n^2)$ for each timestep, and therefore $O((t_{stop} - t_{start})\, n^2)$ for all timesteps.

4 Timed Grammars

Systems are frequently modeled by notion of events which happen at a particular point in time. Typically, these events are atomic, which means, that they do not consume time by themselves. An action, on the other side, starts at some time and ends some time later. Timed grammars are a way to specify how actions are related to another. They are based on the assumption that the observed behavior of some systems can be represented as a hierarchy of actions. A timed grammar includes any possible behavior of a system.

An action is represented as a symbol in the grammar. It can be atomic or composed of other actions. For example, the action “doubleclick” is composed of one click followed by a short break followed by another click. A timed grammar is a set of productions where each production shows how an action on the left side is composed of actions on the right side. Therefore, the expression on the right side shows how one action is related to another action. This is done using an operator symbol.

There are three kinds of relationships between two actions, and one unary modifier operator. Two actions can appear subsequently or concurrently. Also, it is possible, that one action depends on another action. Another operator does not indicate a relationship between two different actions, but modifies one action.

A timed grammar is context-free and hierarchic. A symbol must not evaluate to itself (or to a part of itself), neither directly nor indirectly.

A natural example includes modeling the timing within a time-triggered protocol. The actions are hierarchical. A message is part of a slot, a cycle consists of a sequence of slots and at the top-level the runtime of the network consists of subsequent cycles. The durations of slots depend on the minimum and maximum durations of messages.

The term *action* is discussed in the subsequent section. Section 4.2 gives a formal definition of a timed grammar and introduces some abbreviations and rules to simplify the notation. Then, the semantics of the operators are explained and an algorithm for deriving timings from a timed grammar is presented.

4.1 Actions

An atomic action is not composed of other actions, and therefore appears as terminal symbol in the grammar. It has a start time and an end time. The timing informations are not part of the grammar, but given separately. Since actions emphasize on the length of a time interval, and not on the absolute time of start and end point, they are defined by *offset* and *duration*. The offset is the time that must pass until the action starts since, for example the end of the beforegoing action. The duration is the time that is needed until the action has ended. The sum of offset and duration is the length of an action.

Non-atomic actions are composed of other atomic or non-atomic actions. The offset and duration of them depends on the kind of the composition, which are expressed through operators in the grammar. For example, the length of a doubleclick is the sum of the length of the first and the second click, and the break between the two clicks. This is represented by a sequencing operator in the grammar. The conclusion is, that the offset and durations of all actions in a timed grammar can be derived, if the offsets and durations of the atomic actions are known. This property of timed grammars can be used to calculate all timing parameters.

The view $\langle a \rangle$ on an action a is the tuple $(\Phi\ \langle a \rangle, \Delta\ \langle a \rangle)$, where $\Phi\ \langle a \rangle$ is the offset, and $\Delta\ \langle a \rangle$ the duration of action a. But if a grammar contains multiple productions for the same non-terminal symbol, there are multiple solutions possible for the respective action. Therefore, there is an entity $\langle\langle a \rangle\rangle$ for each action a, which contains all possible views on the action.

4.1.1 Inexact Timing

The time it takes to perform an action can vary for even the simplest components. If a signal is sent to a bus in a network, it is in general not possible to predict the exact time it takes to arrive at the receivers. But in most cases it is possible to provide a lower and an upper limit. Section 2.6 introduced also the problem of clock synchronization. One result was that two clocks can never be synchronized perfectly. Therefore, it makes sense to define the values of duration and offset of actions as time intervals. This means, the duration or offset of an action does not consume a specific amount of time. Instead, the time it takes to perform an action may be estimated in a range from a minimum to a maximum amount of time.

For example, the minimum amount of time for a transmission of a signal over a network may be 1 ns and the maximum amount of time 5 ns. Then, the action *transmission* has a duration of $\Delta \langle transmission \rangle = [1\,\text{ns}; 5\,\text{ns}]$. Both duration and offset of a view are given as time intervals for any action, but it is possible, that the lower and upper bound match.

A view is defined as follows:

Definition 6 View
A view $\langle a \rangle$ is a tuple $(\Phi \langle a \rangle, \Delta \langle a \rangle)$, where $\Phi \langle a \rangle = [f_1, f_2]$ and $\Delta \langle a \rangle = [d_1, d_2]$ are intervals with $f_1, f_2, d_1, d_2 \in \mathbb{R}^+$.

Φ and Δ represent the offset and the duration of the action, respectively. The length of an action is the sum of offset and duration (see also the rules for calculation with intervals in section 4.3.1).

Definition 7 Length
The length of an action a is the time interval $\Lambda \langle a \rangle = \Phi \langle a \rangle + \Delta \langle a \rangle = [f_1 + d_1, f_2 + d_2]$.

As stated out above, it is possible to have multiple productions for the same action. The consequence is, that there can be multiple views for the same action. The set of all views on the same activity is an entity.

In the example of a network, the channels may behave faulty so that some receivers may receive the current transmission while others don't. So there are two productions for the non-terminal representing the transmission. The first represents the empty transmission (maybe a timeout), the second the valid transmission composed of a delay and a subsequent message. The two solutions derived from these productions are both possible for both receiving actions at the receivers. Each action in a timed grammar is associated with one entity with at least one view.

Definition 8 ***Entity***
If e is an action, then $\langle\langle e \rangle\rangle$ is an entity. An entity $\langle\langle e \rangle\rangle$ is a set of views $\langle\langle e \rangle\rangle = \{\langle e \rangle_1, \langle e \rangle_2, \ldots, \langle e \rangle_n\}$ and $\langle\langle e \rangle\rangle \neq \emptyset$.

Throughout the rest of this document, the following notational conventions should be used: If $b = [a_1; a_2]$, then $\min b = a_1$ and $\max b = a_2$. Furthermore, if $\min b = \max b = a$, then the value of b can be written as $[a]$, so that $b = [a] = [a; a]$.

4.1.2 Example: Clock Units

This example introduces the idea of *clock units*, which are the basis of timing in systems with a distributed time base which has been discussed in detail in section 2.6. A clock unit is the time between two clock ticks of a local clock in a distributed system. In general, this time varies from clock to clock.

In a system with n clocks each clock i has a guaranteed minimum and maximum deviation from the nominal clock rate of $r_{min,i}$ and $r_{max,i}$. The definition of clock units uses the most pessimistic assumption, which means that the difference between maximum and minimum deviation of all clocks is maximum. These are the system wide r_{min} with $r_{min} = \min\{r_{min,1}, \ldots, r_{min,n}\}$ and r_{max} with $r_{max} = \max\{r_{max,1}, \ldots, r_{max,n}\}$.

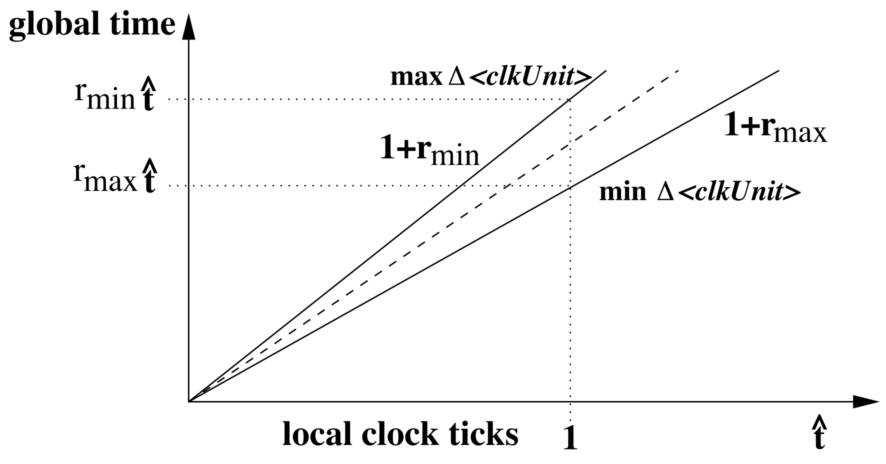


Figure 4.1: The interval between one clock tick $\hat{t}$ of the fastest clock f (with deviation r_{min}) and the slowest clock s (with deviation r_{max}) in a system is the duration of one clock unit Δ $\langle$`clkUnit`$\rangle$.

Definition 9 ***Clock Unit***
If a deviation interval $[r_{min}; r_{max}]$ *is given, the bounds of the duration interval of a clock unit are* $\gamma_f = 1 + r_{min}$ *and* $\gamma_s = 1 + r_{max}$. *A clock unit is an action clkUnit with* Δ $\langle$`clkUnit`$\rangle$ $= [\gamma_f\,\hat{t}; \gamma_s\,\hat{t}]$, *where* $\hat{t}$ *is the nominal duration of a clock tick and* Φ $\langle$`clkUnit`$\rangle$ $= [0]$.

Figure 4.1 shows the relation between deviation and clock units. The real-time duration of a clock tick can be given, for example, in units of ns.

The definition above shows that a clock unit has no offset. As a consequence, actions produced by sequences of clock units, have no offset, and clock units cannot be used to build actions with offsets. The solution for this problem is to swap the values for offset and duration of a clock unit. A *clock offset* is an action $clkOfst$ where the duration of the view is set to Δ $\langle$`clkOfst`$\rangle$ $= [0]$, and the offset is set to Φ $\langle$`clkOfst`$\rangle$ $= \Delta$ $\langle$`clkUnit`$\rangle$. Clock offsets can be used to define offsets with the same timing properties as clock units.

4.2 Timed Grammar

A timed grammar is a special case of a context-free grammar. The productions are ordered hierarchically, in the sense that no expression which can be derived from a non-terminal symbol can contain this symbol. This means, that recursion is excluded. To come to a precise definition, the terms *partition* and *dependency* are defined. A partition is the set of all productions with the same non-terminal symbol on the left side. It directly depends on another partition, if the respective non-terminal symbol of the second partition can be found on the right side of at least one production in the first partition. Dependency is the transitive closure of direct dependancy.

Definition 10 Partitions
A partition $part(P, v)$ *(for short* P_v*) is the set of productions out of* P *in an arbitrary context free grammar* $G = (V, T, P, s)$ *and* $v \in V$, *where all productions of the shape* $v \longrightarrow \rho$ *out of* P *are member of* $part(V, v)$ *and* ρ *is an arbitrary right side expression of a production.*
If $v, v' \in V$ *with* $v \neq v'$, *then* $part(P, v) \cap part(P, v') = \emptyset$ *and* $\bigcup_{v \in V} part(P, v) = P$.

Example 1
The following grammar is given:

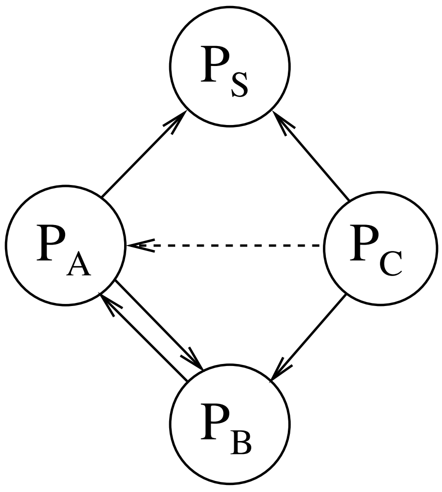


Figure 4.2: Example dependency tree.

p_1: $s \rightarrow a\ c$
p_2: $a \rightarrow b\ a$
p_3: $a \rightarrow b$
p_4: $a \rightarrow \texttt{x}$
p_5: $b \rightarrow a\ c$
p_6: $c \rightarrow \texttt{y}$
p_7: $c \rightarrow \texttt{z}$
where $V = \{a, b, c, s\}$, $T = \{\texttt{x}, \texttt{y}, \texttt{z}\}$ and $P = \{p_1, \ldots, p_7\}$.

The partitions are: $P_s = \{p_1\}$, $P_a = \{p_2, p_3, p_4\}$, $P_b = \{p_5\}$ and $P_c = \{p_6, p_7\}$.

Definition 11 Direct dependency
A partition P_v directly depends on another partition P_w, if there exist a production of the form $v \rightarrow \alpha\ w\ \beta$ in P_v, where α and β are arbitrary expression including the empty expression. The direct dependency relation is written $P_v \lhd P_w$.

Direct dependency applies also to subsets of partitions, if the condition is not violated and both subsets are non-empty.

Definition 12 Dependency
The dependency relation $\lhd^*$ is the transitive closure on the direct dependency relation. This means, that $P_v \lhd^* P_w$, if there is a sequence of direct dependencies from P_v to P_w.

Example 2
In the example above, P_s directly depends on P_a and P_c. P_a directly depends on itself and P_b, while $P_b \lhd P_a$ and $P_b \lhd P_c$. Additionally, $P_a \lhd^* P_c$ because $P_a \lhd P_b$ and $P_b \lhd P_c$. Moreover, $P_b \lhd^* P_b$, because $P_b \lhd P_a$ and $P_a \lhd P_b$. P_c is independent. The dependancies are shown in figure 4.2.

Definition 13 Hierarchical grammar
The grammar is said to be hierarchic, if the following conditions are fulfilled:

1. *For all partitions P_v and P_w with $P_v \neq P_w$ and $P_v \lhd^* P_w$ there is no dependancy $P_w \lhd^* P_v$.*
2. *If s is the start-symbol of the grammar $\mathrm{G} = (V, T, P, s)$, then $P_s \lhd^* P_{v_i}$ for all $v_i \in V$.*

Example 3
The example grammar above violates condition 1, because $P_a \lhd^* P_b$ and $P_b \lhd^* P_a$. Consider a new grammar $\mathrm{G}' = (V, T, P', s)$ where production p_5 is replaced by a production $b \to c$. In this case condition 1 is fulfilled, and furthermore P_s depends on P_a, P_b and P_c (condition 2).

Definition 14 Timed Grammar
The timed grammar TG *is a five-tuple* $\mathrm{TG} = (V, T, O, P, s)$*, with*

V	*a set of syntactical variables*
T	*a set of terminal symbols*
O	*a set of operators:* $O = \{\|, \circ, ., \square\}$
P	*a set of productions*
s	*a variable* $s \in V$

and $\mathrm{G} = (V, T \cup O, P, s)$ *is a hierarchical grammar. Each production in P has one of the following forms:*

$$\begin{aligned} a &\to \mathtt{t} \\ a &\to b \\ a &\to \square\, b \\ a &\to b\; p\; c \end{aligned}$$

where $a, b, c \in V$, with $a \neq b$, $b \neq c$, $a \neq c$, $\mathtt{t} \in T$, and $p \in \{\|, \circ, .\}$.

In the context of timed grammars all symbols in $V \cup T$ represent actions, so that there is one entity $\langle\langle a \rangle\rangle$ for each $a \in V \cup T$ with at least one view $\langle a \rangle \in \langle\langle a \rangle\rangle$.

Because the number of productions in a timed grammar can become very high, operator precedence rules will be introduced, to write more complex expressions in fewer productions. The following table lists the operator precedence from highest to lowest, where α, α_1, α_2 are expressions.

Precedence	Operation
1	(α)
2	$\square\, \alpha$
3	$\alpha_1 \,.\, \alpha_2$
4	$\alpha_1 \circ \alpha_2$
5	$\alpha_1 \,\|\, \alpha_2$

For example, the following production

$$a \to b \circ (c \,\|\, d) \,.\, \square\, e$$

can be turned into a timed grammar TG_1 with $V_1 = V \cup \{v_1, v_2, v_3\}$ in the following way:

$$\begin{aligned} v_1 &\to c \,\|\, d \\ v_2 &\to \square\, e \\ v_3 &\to v_1 \,.\, v_2 \\ a &\to b \circ v_3 \end{aligned}$$

If a symbol appears subsequently a multiple number of times in the right side of a production, then this can be expressed using an ellipsis, for example $a \to b \,.\, \ldots \,.\, b$. The actual number of symbols on the right side is denoted by $|a|$. There should be no other symbol in that production. If, for example, $|a| = 4$, then the production above is $a \to b \,.\, b \,.\, b \,.\, b$. Although, the notation suggests, that there are at least two symbols on the right side, the case $|a| = 1$ should be possible.

Similarly, it should be possible to use the ellipsis to derive words with a minimum and a maximum number of symbols. For example, the words $b \,.\, b \,.\, b$, $b \,.\, b \,.\, b \,.\, b$ and $b \,.\, b \,.\, b \,.\, b \,.\, b$ are in L(TG), but neither $b \,.\, b$ nor $b \,.\, b \,.\, b \,.\, b \,.\, b \,.\, b$. This can be written $|a| = (3, 5)$ to denote that the production $a \to b \,.\, \ldots \,.\, b$ can evaluate to words with three, four or five symbols, but not more or less. It is a shorthand for the following three productions:

$$\begin{aligned} a &\to b \,.\, b \,.\, b \\ a &\to b \,.\, b \,.\, b \,.\, b \\ a &\to b \,.\, b \,.\, b \,.\, b \,.\, b \end{aligned}$$

4.3 Timed Grammars and Entities

The words of the language L(TG) consists of terminal symbols and operators, which describe the relationship between actions. The kind of relationship depends on the operators. In the following, the meaning of the operator symbols will be clarified.

4.3.1 Calculation Rules for Intervals

To build views from other views, four different operators are provided. Using these operators, it is possible to construct a new entity from the views of two different entities. Since the Φ and Δ values of a view are intervals, it is necessary to provide the following frequently used calculation rules on intervals.

$$\begin{aligned} [a_1; a_2] + [b_1; b_2] &= [a_1 + b_1; a_2 + b_2] \\ \min\{[a_1; a_2], [b_1; b_2]\} &= [\min\{a_1, b_1\}; \min\{a_2; b_2\}] \\ \max\{[a_1; a_2], [b_1; b_2]\} &= [\max\{a_1, b_1\}; \max\{a_2; b_2\}] \\ c\,[a; b] &= [c\,a; c\,b] \\ \frac{[a; b]}{c} &= \frac{1}{c}\,[a; b] \\ \frac{c}{[a; b]} &= \left[\frac{c}{a}; \frac{c}{b}\right] \end{aligned}$$

where $a_1, a_2, b_1, b_2, a, b, c \in \mathbb{R}$.

Furthermore,

$$\begin{aligned} [a; b] > [c; d] &\Leftrightarrow a > d \\ [a; b] < [c; d] &\Leftrightarrow b < c \\ [a; b] \geq [c; d] &\Leftrightarrow a \geq d \\ [a; b] \leq [c; d] &\Leftrightarrow b \leq c \end{aligned}$$

4.3.2 Relationship Between Actions

Timed grammars show the relation between time intervals. The terminals represent the “basic blocks” in a system, while each production builds a new time interval depending on the kind of relation. The syntax-tree of a derived

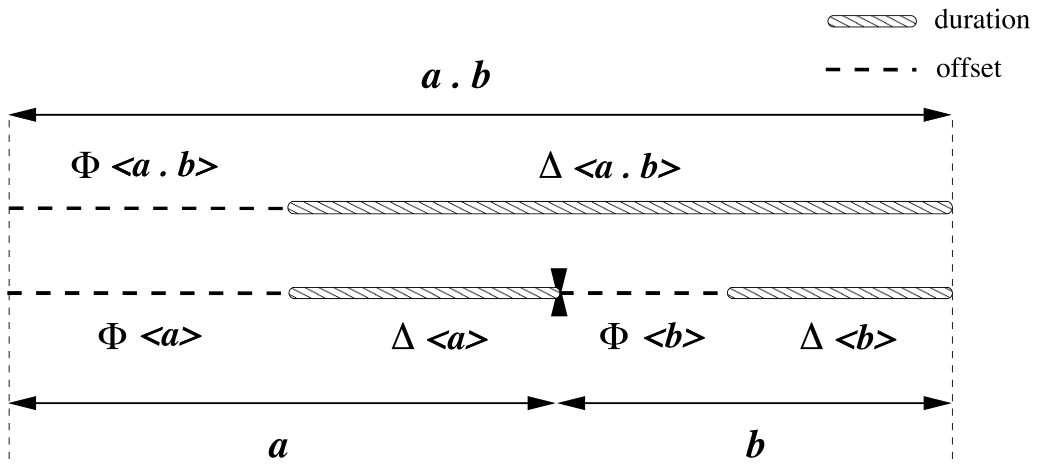


Figure 4.3: An action $a \,.\, b$ composed of two subsequent actions a and b.

word out of L(TG) represents the hierarchy between the building blocks of the system, and the top level interval. Each node in the syntax-tree can be attributed with an entity which encapsulates a set of time intervals. These intervals are the different views from components in a distributed system on the part in the hierarchy which is represented by the grammatical element at that node in the syntax-tree.

The three binary operators ., || and ∘ describe a relationship between two actions. By setting two actions in relation to another, a new activity can be composed. The duration and offset of the view of the composed action is determined depending on the operator. If there are multiple views, then the new entity contains each combination of views of the operands' entities. If α is a right-side expression of a production in a timed grammar, then $\langle\langle\alpha\rangle\rangle$ is the resulting entity of the respective composition according to the rules below. For example, if there is a production $a \to b \,.\, c$, then $\langle\langle b \,.\, c\rangle\rangle$ is the entity which is composed of two subsequent entities $\langle\langle b\rangle\rangle$ and $\langle\langle c\rangle\rangle$.

The remainder of this section gives an overview of these operators, and how they are related to views and entities.

Subsequent Actions

The operator . defines two actions as subsequent, which means that the second action follows the first action in time. More precisely, the expression $a \,.\, b$ means that the offset of b starts at the end of a. Figure 4.3 illustrates the sequence operator. If two actions a and b are subsequent, and $\langle\langle a\rangle\rangle$ and $\langle\langle b\rangle\rangle$ are the respective entities, then for all $i \in \{1,\ldots,|\langle\langle a\rangle\rangle|\}$, $j \in \{1,\ldots,|\langle\langle b\rangle\rangle|\}$ and $k = |\langle\langle a\rangle\rangle|(j-1)+i$:

$$\langle\langle a \,.\, b\rangle\rangle = \bigcup \{(\Phi\,\langle c\rangle_k\,,\Delta\,\langle c\rangle_k)\}$$

with

$$\begin{aligned}\Phi\,\langle c\rangle_k &= \Phi\,\langle a\rangle_i\\ \Delta\,\langle c\rangle_k &= \Delta\,\langle a\rangle_i + \Phi\,\langle b\rangle_j + \Delta\,\langle b\rangle_j\end{aligned}$$

if $\Delta\,\langle a\rangle_i \neq 0$, or

$$\begin{aligned}\Phi\,\langle c\rangle_k &= \Phi\,\langle a\rangle_i + \Phi\,\langle b\rangle_j\\ \Delta\,\langle c\rangle_k &= \Delta\,\langle b\rangle_j\end{aligned}$$

otherwise.

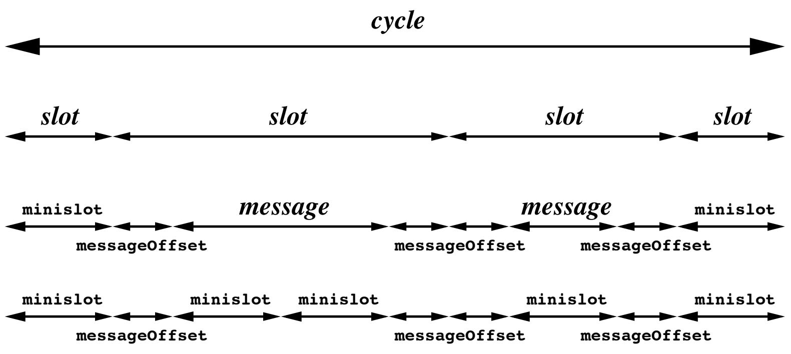


Figure 4.4: A cycle is built up by consequtive slots.

The following example grammar shows how the sequence operator can be used to describe a very simple minislot arbitration scheme (see also section 2.4.3).

$$\begin{aligned} cycle &\rightarrow slot\,.\,\ldots\,.\,slot \\ slot &\rightarrow \texttt{minislot} \\ slot &\rightarrow \texttt{messageOffset}\,.\,message\,.\,\texttt{messageOffset} \\ message &\rightarrow \texttt{minislot}\,.\,\ldots\,.\,\texttt{minislot} \end{aligned}$$

A cycle is a sequence of slots, which can be either one minislot (in case that the currently respective owner of the slot does not send) or it is filled with a message. A sender has to wait a short amount of time, which is represented by the terminal `messageOffset`. In this example, a message extends over multiple minislots. The byteflight protocol uses a similar scheme.

The grammar uses the ellipsis notation two twice. The first production requires the information about how many slots are required for a cycle. In this example there should be four slots per cycle, so that $|cycle| = 4$. This is equivalent to $cycle \rightarrow slot\,.\,slot\,.\,slot\,.\,slot$. The minimum length of a message should be one and the maximum length three minislots, so that $|message| = (1, 3)$.

Figure 4.4 shows one possible example situation. The respective word is derived from the grammar in the following way:

$$\begin{aligned} cycle &\mapsto slot\,.\,slot\,.\,slot\,.\,slot \\ &\mapsto \texttt{minislot}\,.\,\texttt{messageOffset}\,.\,message\,.\,\texttt{messageOffset}\,.\,\texttt{messageOffset} \\ &\quad .\,message\,.\,\texttt{messageOffset}\,.\,\texttt{minislot} \\ &\mapsto .\,\texttt{minislot}\,.\,\texttt{messageOffset}\,.\,\texttt{minislot}\,.\,\texttt{minislot}\,.\,\texttt{messageOffset} \\ &\quad .\,\texttt{messageOffset}\,.\,\texttt{minislot}\,.\,\texttt{messageOffset}\,.\,\texttt{minislot} \end{aligned}$$

This word represents a situation where the owners of slot two and three are sending a message of the length two and one minislot durations.

Concurrent Actions

The || operator symbol is used if two actions run concurrently. The result of $a \parallel b$ is an action with the maximum length of both operands a and b (see figure 4.5), so that

$$\begin{aligned} \Phi\,\langle c\rangle_k &= \min\{\Phi\,\langle a\rangle_i\,, \Phi\,\langle b\rangle_j\} \\ \Delta\,\langle c\rangle_k &= \max\{0, \max\{\Lambda\,\langle a\rangle\,, \Lambda\,\langle b\rangle\} - \Phi\,\langle c\rangle_k\} \\ \langle\langle a \parallel b\rangle\rangle &= \bigcup \{(\Phi\,\langle c\rangle_k\,, \Delta\,\langle c\rangle_k)\} \end{aligned}$$

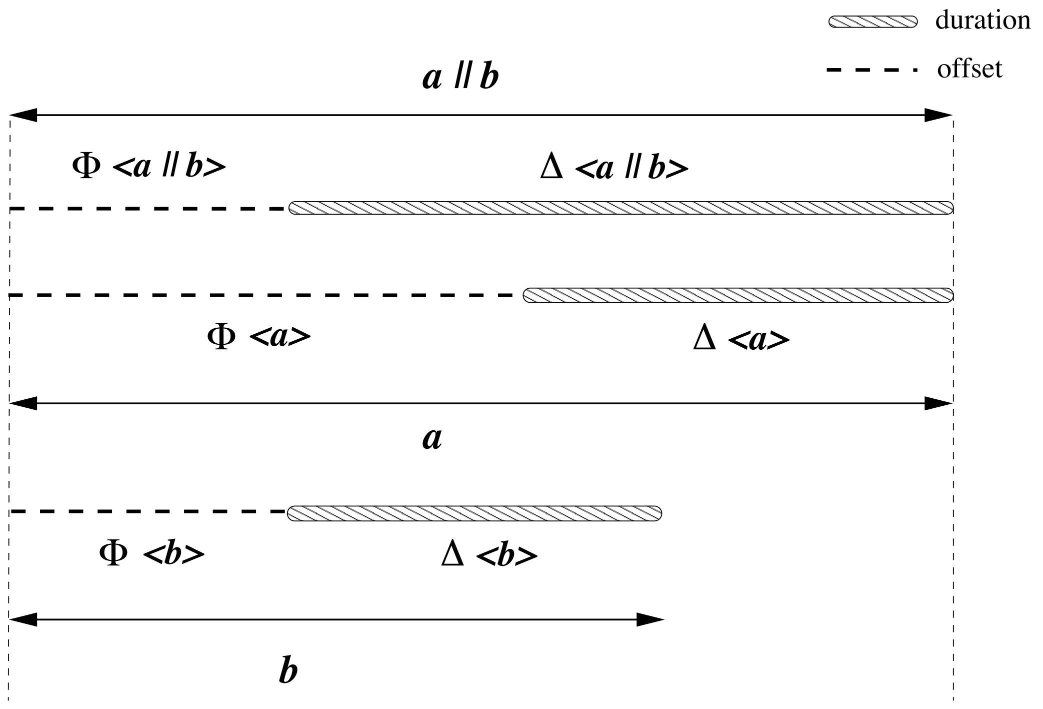


Figure 4.5: An action $a \parallel b$ composed of two concurrent actions a and b.

for all $i \in \{1, \ldots, |\langle\langle a \rangle\rangle|\}$, $j \in \{1, \ldots, |\langle\langle b \rangle\rangle|\}$ and $k = |\langle\langle a \rangle\rangle|(j-1) + i$.

One possible application example is the `fork()`/`wait()` mechanism found in most modern operating systems. The `fork()` system call spawns a new process and returns the process id of the newly created process to the caller (the parent process), while returning 0 to the child process. Both processes run concurrently, until either the parent process calls `wait()` or the child process terminates. The execution of the respective other process continues, until both conditions are fulfilled.

The following algorithm shows an example. It uses `waitpid()` instead of `wait()`, to wait for processes with a certain process id. Although, this is not strictly necessary, it is a way to distinct the first child (with its process id stored in `pid1`) from all other childs. Figure 4.6 illustrates the run-time behavior of the algorithm.

```
if $n \le 0$ or $n >$ MAX then
   print "error: n must be greater than 0"
   exit(FAILURE)
else
   $pid1 \leftarrow$ fork()
   if $pid1 = 0$ then
      sub1()
      exit(SUCCESS)
   else
      for $i = 0$ to $n$ do
         $tmp \leftarrow$ fork()
         if $tmp = 0$ then
            sub2($i$)
            exit(SUCCESS)
```

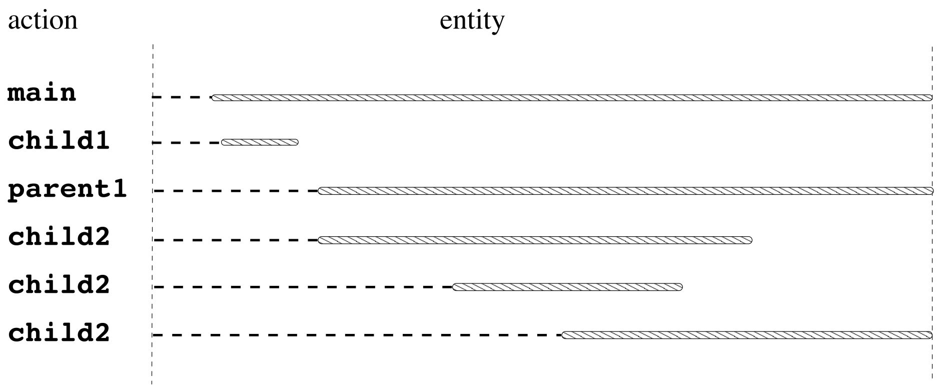


Figure 4.6: A possible timing in the `fork()/wait()` scenario.

```
15:         else
16:             $pid[i] \leftarrow tmp$
17:         end if
18:     end for
19:     for $i = 0$ to $n$ do
20:         waitpid($pid[i]$)
21:     end for
22:     waitpid($pid1$)
23:   end if
24: end if
25: return SUCCESS
```

A timed grammar which models the behavior of the program is

$$\begin{aligned}
main &\rightarrow (parent1 \parallel child1)\,.\,\texttt{returnSuccess}\\
main &\rightarrow \texttt{exitFailure}\\
child1 &\rightarrow \texttt{sub1}\,.\,\texttt{exitSuccess}\\
parent1 &\rightarrow child2 \parallel \ldots \parallel child2\\
child2 &\rightarrow \texttt{sub2}\,.\,\texttt{exitSuccess}
\end{aligned}$$

where $|parent1| = (1, \texttt{MAX})$. The main program can either fail, or spawn a child process, so that parent (action *parent*1) and its child (action *child*1) run in parallel. The child ends at line 8 and the parent waits for the child at line 22, unless the parent reaches line 22 earlier. In this case it is suspended until the child exits. In the meanwhile, the parent spawns n childs each calling a procedure `sub2()` which takes an argument i. These calls are executed concurrently.

The example shows, that the length of the resulting action in every step is the length of the slowest action. Therefore, the offset and duration of the result is the minimum offset and the maximum duration of the arguments.

Actions as Limits for other Actions

In contrast to the concurrency operator, the $\circ$ operator takes the minimum duration and the maximum offset. The duration of the result is the part of the duration of the first operand which overlaps with the duration of the second operand, so that

$$\begin{aligned}
\Phi \langle c \rangle_k &= \max\{\Phi \langle a \rangle_i , \Phi \langle b \rangle_j\} \\
\Delta \langle c \rangle_k &= \max\{0, \min\{\Lambda \langle a \rangle , \Lambda \langle b \rangle\} - \Phi \langle c \rangle_k\} \\
\langle\langle a \circ b \rangle\rangle &= \bigcup \{(\Phi \langle c \rangle_k , \Delta \langle c \rangle_k)\}
\end{aligned}$$

for all $i \in \{1, \ldots, |\langle\langle a \rangle\rangle|\}$, $j \in \{1, \ldots, |\langle\langle b \rangle\rangle|\}$ and $k = |\langle\langle a \rangle\rangle|(j - 1) + i$.

Timeouts are one example for this operator. Figure 4.7 illustrates a scenario where a client sends a request to a server and then waiting for a response. The time the client waits is limited by a timeout. The following grammar shows a production for the clients action.

$$client \rightarrow request\,.\,(response \circ \texttt{timeout})$$

Subsuming Views

The unary □-operator modifies the duration and length of all views in the entity. The offset of $\Phi \langle \square\, a \rangle$ is set to 0, while $\Delta \langle \square\, a \rangle = \Phi \langle a \rangle + \Delta \langle a \rangle$. It can be used in connection with other operators. If, in the example above, the *timeout* action has an offset $\Phi \langle timeout \rangle > 0$, the production may lead to unexpected results, if $\Phi \langle response \rangle < \Phi \langle timeout \rangle$, because parts of the duration of $\langle response \rangle$ can be cut-off at the beginning (see figure 4.8). If, on the other side, □ *timeout* is used, then the result of $\Phi \langle response \circ \square\, timeout \rangle$ is always $\max\{\Phi \langle response \rangle , 0\} = \Phi \langle response \rangle$.

4.3.3 Multiple Solutions

If a partition has more than one production, then there can be multiple solutions for the duration and offset for that action and the actions depending on this action. If ρ_1 and ρ_2 are right side expressions for the same non-terminal a, and the following productions given:

$$\begin{aligned}
a &\rightarrow \rho_1 \\
a &\rightarrow \rho_2
\end{aligned}$$

then $\langle\langle a \rangle\rangle = \langle\langle \rho_1 \rangle\rangle \cup \langle\langle \rho_2 \rangle\rangle$.

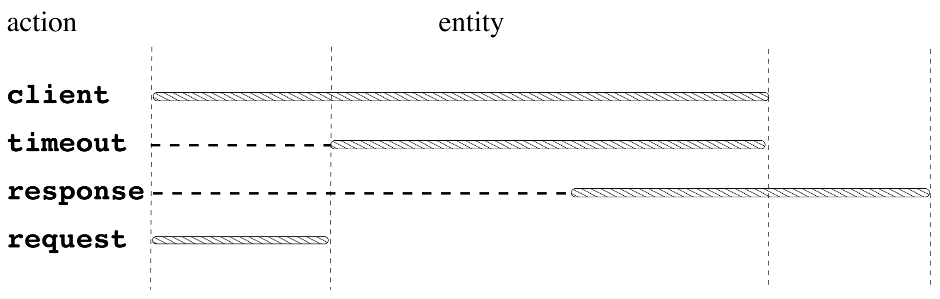


Figure 4.7: A client sends a request for a server, waiting for response or a timeout. In this example, the timeout expires during the reply of the server. Therefore, the clients action ends with the timeout.

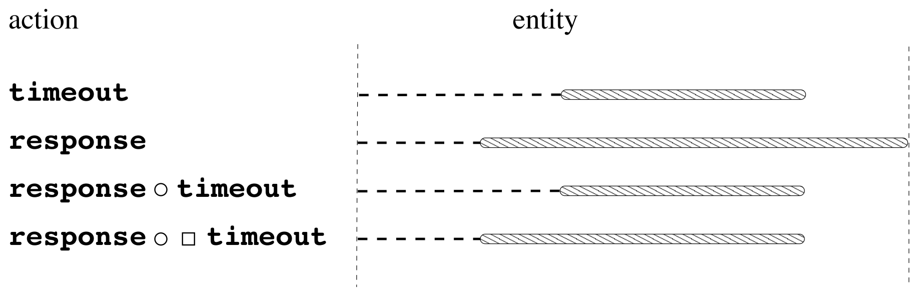


Figure 4.8: In this example, the □-operator assures, that the duration of the *response* action is not cut-off at the beginning.

The following example shows, how multiple productions for one action lead to entities with more than one view. The grammar shows a possible realization of the scenario mentioned in section 4.1.1 where a transmission can be either a message or there can be a missing message, for example in the case of a transmission fault.

$$\begin{aligned}
transmission &\rightarrow \texttt{messageMissing}\\
transmission &\rightarrow \texttt{msg}\\
receiver1 &\rightarrow receiption\\
receiver2 &\rightarrow receiption\\
receiption &\rightarrow receiver1 \parallel receiver2
\end{aligned}$$

where *receiption* is the start symbol. Entities with multiple views can exist only, if there is at least one partition with more than one production. This is the case for $part(P, transmission)$ where there are 2 productions.

The words which can be derived from the grammar are:

$$\begin{aligned}
receiption &\Rightarrow \texttt{messageMissing} \parallel \texttt{msg}\\
receiption &\Rightarrow \texttt{msg} \parallel \texttt{messageMissing}\\
receiption &\Rightarrow \texttt{messageMissing} \parallel \texttt{messageMissing}\\
receiption &\Rightarrow \texttt{msg} \parallel \texttt{msg}
\end{aligned}$$

and the respective views in $\langle\langle receiption\rangle\rangle$:

$$\begin{aligned}
\Phi\,\langle receiption\rangle_1 &= \min\{\Phi\,\langle\texttt{messageMissing}\rangle, \Phi\,\langle\texttt{msg}\rangle\}\\
\Delta\,\langle receiption\rangle_1 &= \max\{0, \max\{\Lambda\,\langle\texttt{messageMissing}\rangle, \Lambda\,\langle\texttt{msg}\rangle\} - \Phi\,\langle receiption\rangle_1\}\\
\Phi\,\langle receiption\rangle_2 &= \min\{\Phi\,\langle\texttt{msg}\rangle, \Phi\,\langle\texttt{messageMissing}\rangle\}\\
\Delta\,\langle receiption\rangle_2 &= \max\{0, \max\{\Lambda\,\langle\texttt{msg}\rangle, \Lambda\,\langle\texttt{messageMissing}\rangle\} - \Phi\,\langle receiption\rangle_2\}\\
\Phi\,\langle receiption\rangle_3 &= \min\{\Phi\,\langle\texttt{messageMissing}\rangle, \Phi\,\langle\texttt{messageMissing}\rangle\}\\
\Delta\,\langle receiption\rangle_3 &= \max\{0, \max\{\Lambda\,\langle\texttt{messageMissing}\rangle, \Lambda\,\langle\texttt{messageMissing}\rangle\} - \Phi\,\langle receiption\rangle_3\}\\
\Phi\,\langle receiption\rangle_4 &= \min\{\Phi\,\langle\texttt{msg}\rangle, \Phi\,\langle\texttt{msg}\rangle\}\\
\Delta\,\langle receiption\rangle_4 &= \max\{0, \max\{\Lambda\,\langle\texttt{msg}\rangle, \Lambda\,\langle\texttt{msg}\rangle\} - \Phi\,\langle receiption\rangle_4\}
\end{aligned}$$

where $\langle\langle\texttt{messageMissing}\rangle\rangle = \{\langle\texttt{messageMissing}\rangle\}$ and $\langle\langle\texttt{msg}\rangle\rangle = \{\langle\texttt{msg}\rangle\}$.

Three different solutions are possible: $\langle receiption\rangle_1$, $\langle receiption\rangle_3$ and $\langle receiption\rangle_4$, and therefore

$$\langle\langle receiption\rangle\rangle = \{\langle receiption\rangle_1, \langle receiption\rangle_3, \langle receiption\rangle_4\}$$

$\langle receiption\rangle_2$ does not appear in $\langle\langle receiption\rangle\rangle$, because $\langle receiption\rangle_1 = \langle receiption\rangle_2$.

4.4 Evaluation of Timed Grammars

The relationship between timed grammars and entities makes it possible to automatically calculate the values of entities and views for actions, if a timed grammar and the entities for the terminal symbols in the grammar, are given. The following algorithm shows how all entities can be calculated.

The algorithm uses different data types for entities, views, partitions, productions, etc. A value of type $Symbol$ is a literal, which represents a terminal or non-terminal symbol of a timed grammar. This type is used to define the abstract data type *Grammatical_Expression*, which represents the right side expression of a production (see below).

ADT *Grammatical_Expression*

constructors :

- $replacement : Symbol \mapsto Grammatical\_Expression$
  - An expression of the type a, where $a \in V \cup T$
- $subsumption : Symbol \mapsto Grammatical\_Expression$
  - An expression of the type $\Box\, a$
- $sequence, concurrency, limit : Symbol \times Symbol \mapsto Grammatical\_Expression$
  - An expression of the type $a\ .\ b$ (sequence), $a \parallel b$ (concurrency) or $a \circ b$ (limit)

selectors :

- $action : Grammatical\_Expression \mapsto Symbol$
  - $action(replacement(a)) = a$
  - $action(subsumption(a)) = a$
  - $action(\rho)$ is undefined in any other case
- $action_1 : Grammatical\_Expression \mapsto Symbol$
  - $action_1(sequence(a, b)) = a$
  - $action_1(concurrency(a, b)) = a$
  - $action_1(limit(a, b)) = a$
  - $action_1(\rho)$ is undefined in any other case
- $action_2 : Grammatical\_Expression \mapsto Symbol$
  - $action_2(sequence(a, b)) = b$
  - $action_2(concurrency(a, b)) = b$
  - $action_2(limit(a, b)) = b$
  - $action_2(\rho)$ is undefined in any other case

predicates :

- $is\_replacement, is\_subsumption : Grammatical\_Expression \mapsto \mathbf{boolean}$
  - $is\_replacement(replacement(a)) = \mathbf{true}$
  - $is\_replacement(\rho) = \mathbf{false}$, if $\rho \neq replacement(a)$
  - $is\_subsumption(subsumption(a)) = \mathbf{true}$
  - $is\_subsumption(\rho) = \mathbf{false}$, if $\rho \neq subsumption(a)$
- $is\_sequence, is\_concurrency, is\_limit : Grammatical\_Expression \mapsto \mathbf{boolean}$
  - $is\_sequence(sequence(a, b)) = \mathbf{true}$
  - $is\_sequence(\rho) = \mathbf{false}$, if $\rho \neq sequence(a, b)$
  - $is\_concurrency(concurrency(a, b)) = \mathbf{true}$
  - $is\_concurrency(\rho) = \mathbf{false}$, if $\rho \neq concurrency(a, b)$
  - $is\_limit(limit(a, b)) = \mathbf{true}$
  - $is\_limit(\rho) = \mathbf{false}$, if $\rho \neq limit(a, b)$

A value of the type *Production* consists of a left-side symbol, and a right-side expression.

ADT *Production*

constructor :

$make\_production : Symbol \times Grammatical\_Expression \mapsto Production$
$make\_production(a, \rho)$ represents the production $a \to \rho$

selectors :

$target : Production \mapsto Symbol$
$target(make\_production(a, \rho)) = a$
$right\_side\_expression : Production \mapsto Grammatical\_Expression$
$right\_side\_expression(make\_production(a, \rho)) = \rho$

The algorithm uses a set of partitions, to decent recursively from the partition of the start symbol, to the partitions of the terminal symbols. A partition is a set of productions where all productions have the same left-side symbol.

type *Partition* = **Set of** *Production*

The following can be assumed:

$$\forall \; p_i, p_j \in P_a : target(p_i) = target(p_j) = a$$

`target` : $Partition \mapsto Symbol$ is the function which returns the target of the productions in partition P_a, so that `target`$(P_a) = a$.

Furthermore, there is a type *Interval*, with all operators available as shown in section 4.3.1 and a type *Entity*. The abstract data type for views is shown below:

ADT *View*

constructor :

$make\_view : Interval \times Interval \mapsto View$
$make\_view(\Phi \langle a \rangle, \Delta \langle a \rangle) = \langle a \rangle$

selectors :

$offset : View \mapsto Interval$
$offset(make\_view(\Phi \langle a \rangle, \Delta \langle a \rangle)) = \Phi \langle a \rangle$
$duration : View \mapsto Interval$
$duration(make\_view(\Phi \langle a \rangle, \Delta \langle a \rangle)) = \Delta \langle a \rangle$

type *Entity* = **Set of** *View*

The operators □, ., || and ∘ are defined for entities as shown in section 4.3.2.

The function `evaluate_expression` returns a new entity by evaluating the right-side expression of a production. It takes a dictionary `Entities`, which maps the actions to the entities already known. It is initialised with the atomic actions, which are given as input to the algorithm. During runtime, the dictionary is filled up with the results of the evaluation of the single partitions. An action, whose entity is not yet available (which means it has to be calculated) is mapped to **nil**. If the entity of an action in a grammatical expression is not known, then `evaluate_partition` is invoked with the partition of the respective action. This function is shown later in this section. Evaluation of an expression requires the knowledge of the entities of the argument actions. For example, if $\rho = a \; . \; b$, then $\langle\langle a \rangle\rangle$ and $\langle\langle b \rangle\rangle$ must be known in order to return $\langle\langle \rho \rangle\rangle = \langle\langle a \; . \; b \rangle\rangle$. This is what this function assures.

```
1: function evaluate_expression(Map (Symbol, Entity) Entities; Grammatical_Expression $\rho$) returns
   Set of View
2: Set of View result $\leftarrow \emptyset$
3: if $is\_replacement(\rho) \lor is\_subsumption(\rho)$ then
4:     $a \leftarrow action(\rho)$
5:     if Entities[$a$] = nil then
6:         $\langle\langle a \rangle\rangle \leftarrow$ evaluate_partition(Entities, $P_a$)
7:     else
8:         $\langle\langle a \rangle\rangle \leftarrow$ Entities[$a$]
```

```
 9:     **end if**
10:     $result \leftarrow \langle\langle\rho\rangle\rangle$
11: **else**
12:     $a \leftarrow action_1(\rho)$
13:     $b \leftarrow action_2(\rho)$
14:     **if** Entities[$a$] = **nil then**
15:         $\langle\langle a\rangle\rangle \leftarrow$ evaluate_partition(Entities, $P_a$)
16:     **else**
17:         $\langle\langle a\rangle\rangle \leftarrow$ Entities[$a$]
18:     **end if**
19:     **if** Entities[$b$] = **nil then**
20:         $\langle\langle b\rangle\rangle \leftarrow$ evaluate_partition(Entities, $P_b$)
21:     **else**
22:         $\langle\langle b\rangle\rangle \leftarrow$ Entities[$b$]
23:     **end if**
24:     $result \leftarrow \langle\langle\rho\rangle\rangle$
25: **end if**
26: **return** $result$
27: **end function**
```

The resulting entity of a partition containing more than one production is the union set of all results of the `evaluate_expression()` calls for each production in the respective partition. The following function sums up all views of one partition.

```
 1: **function** evaluate_partition(**in out Map** ($Symbol$, $Entity$) Entities; $PartitionP_a$) **returns** $Entity$
 2: **Set of** $View$ $partitions\_result \leftarrow \emptyset$
 3: **for all** $p \in P_a$ **do**
 4:     $\rho \leftarrow right\_side\_expression(p)$
 5:     $partitions\_result \leftarrow partitions\_result \cup$ evaluate_expression(Entities, $\rho$)
 6: **end for**
 7: $a \leftarrow$ target($P_a$)
 8: Entities[$a$] $\leftarrow partitions\_result$
 9: **return** $partitions\_result$
10: **end function**
```

Note, that the function asynchronously updates the set `Entities`. The algorithm starts with adding all entities of the atomic actions to `Entities` and applying the function `evaluate_partition()` to the partition P_s of the start symbol s.

```
1: **Map** ($Symbol$, $Entity$) Entities
2: **for all** t $\in T$ **do**
3:     Entities[t] $\leftarrow \langle\langle$t$\rangle\rangle$
4: **end for**
5: $\langle\langle s\rangle\rangle$ = evaluate_partition(Entities, $P_s$)
6: // Entities has been asynchronously updated, and contains all entities, now.
```

Part II

The TEA Protocol

5 Basic Operation

5.1 Overview: Faults to be Tolerated

A fault-tolerant protocol should be able to protect ongoing communication in case of faults. The main source for faulty behavior (amongst others) are the channel and the controllers, because they are the most important components in every network. This section summarizes possible behavior of faulty controllers and channels. In addition, possible faulty behavior of additional components must be taken into account, depending on the architecture. For the TEA network architecture, this topic will be discussed in 7.

1. Faulty controller:
   Besides behaving fault-free, a faulty controller may cause different kinds of malfunction.
   - A controller can encode wrong values in a message. These kind of faults can be easily detected, if the sent value is not valid, for example in case of a boundary violation. If this is not the case, it is not possible to come to a correct solution without additional information from other (fault-free) devices.
   - Physical encodings of the message may be incorrect. A signal may be malformed, so that it does not meet the specification, for example with respect to the bit-timing and incorrect signal levels. In TEA it is required, that this kind of message corruption causes symmetric failures (see section 2.2, which means that all receivers accept or reject a message in the same way on the channel the controller sent to.
   - Timing faults mean, that a controller acts at the wrong time. In case of channel access, for example, this may cause collisions and message corruption.
   - A controller may control additional devices for the network to function. All the problems mentioned so far (incorrect value, encoding and timing) can apply in this case and must be taken into account.
   - A controller can behave differently on different channels.
   - Services for the host controller may be denied. Though this is an important problem, the higher levels of communications are not part of this discussion and therefore excluded.
2. Faulty channel:
   The channel can corrupt correct messages and spontaneously generate random signals. In particular, the following things may happen:
   - The channel generates channel noise. In principal, a channel may possibly generate less complex communication tokens. On the other side it is assumed, that the probability for the spontaneous generation of complex frame formats with reasonable protection mechanisms is extremely low. So this problem will not be taken into account any further.
   - Channel noise can corrupt correct messages.
   - Byzantine behavior occurs, if a channel delivers a correct message to only a subset of all receivers.
   - As a special case of the scenarios above, a channel may fail-silent completely.
3. Combined controller and channel faults may happen. All combinations of the above are possible.

One possible fault is left out: A message with a malformed physical encoding, for example sent by a faulty controller, should not become "repaired by accident" by a channel fault. Corrupting a corrupted message in a way, that the result is non-corrupted introduces no new aspects. This results in fault-free behavior, which is possible even in the case of faulty components. In connection with a star- or mixed topology, the corrupted message may be delivered to single receivers, if the fault occurs on single links. This may be single receiver-to-star links, or links from one subnet to another subnet (net-splits). This results in byzantine behavior. All of these situations are covered in the list above.

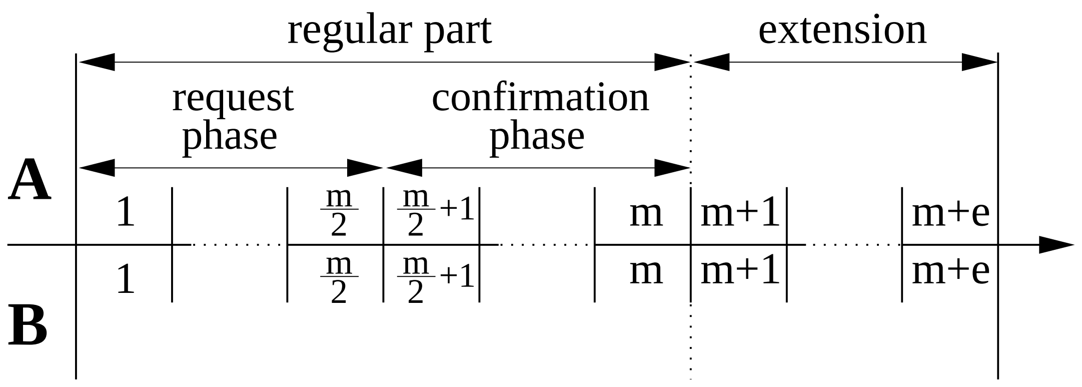


Figure 5.1: The basic structure of a TEA communication cycle with *m* controllers and *e* slots in the extension.

5.2 The TEA Solution

The TEA protocol aims to solve the problems discussed above. Besides a static schedule it provides dynamic access for the controllers to the channel. It is guaranteed that a faulty controller cannot monopolize the channels, so that other controllers can still communicate. In contrast to most other real-time fieldbus protocols, the TEA protocol is designed to tolerate double-faults. This means, that simultaneous faults in faulty controller and one channel can be tolerated.

TEA uses a double-controller approach to protect the channels against collisions. When a sender wants to access a channel it must be granted by the neighbor controller on the same node. If the neighbor becomes a sender in another slot or one the other channel in the same slot, the former sender is now responsible for access control of the former neighbor (see chapter 6).

To guarantee fault-tolerant dynamic arbitration, TEA reserves a part of the bandwidth which can be used for dynamic allocation. The schedule of senders for this extension part is pre-built at the end of the preceding regular part, which itself has a fixed schedule. Figure 5.1 shows the structure of a communication cycle. The messages include protocol-relevant informations. The payload data is placed at the end of the message. Each controller sends a request bit (which is set if the controller requests a slot) in the first half of the regular part (request phase). Then all controllers send the whole vector of all received request bits in the second half of the regular part (confirmation phase). Even in the case of message corruption and byzantine faults each controller has received enough information to build a schedule for the extension part. Therefore, an agreement algorithm is applied on the information which is available locally, so that all controllers come to the same result. This happens just before the start of the extension part. The agreement algorithm is subject to chapter 8.

This approach has some advantages. First, the next sender is already known by each controller, especially by the neighbor controller of the sender which has to allow access to the channels. Senders are not determined by only listening, so that no extra time is needed to wait for possible signals on the channels from other senders, which would be necessary otherwise. The channels remain protected even in the case of faulty controllers. Second, it is not necessary to wait several timeouts to prevent collisions. This saves bandwidth and the extension part is used more efficiently. A previously known schedule supports also preserving fault-tolerance in the case of non-static message length. The problem of messages with dynamic length is postponed to chapter 11.

Some topics remain open. The protocol focusses on the “normal operation” part of communication. This excludes activities like protocol startup and shutdown. TEA also uses a global distributed time base, but makes no assumptions about the kind of clock synchronization, except that all clocks of non-faulty controllers are synchronized with a reasonable limited clock skew. Clock synchronization itself is not part of TEA.

TEA provides solutions for all faults listed in section 5.1. Subsequent chapters will go into details. The following list gives an overview over the fault-tolerance strategies in TEA:

1. Faulty controller:
   - The agreement algorithm provides fault-tolerance against inconsistent or non-available requests from faulty controllers.
   - Two controllers on one node guard each other, so that timing faults are prevented effectively.
2. Faulty channel:
   - The agreement algorithm tolerates also byzantine faults caused by a faulty channel.
   - Message protection mechanisms like checkword mechanisms, provide protection against value faults caused by message corruption.
   - TEA uses a double-channel architecture, so that the fault-free controllers can always communicate over the fault-free channel.
3. The fault-tolerance mechanisms make it possible to tolerate one faulty channel and one faulty controller at the same time.

5.3 Basic Protocol Operation

This section describes the structure of the protocol as timed grammar.

The protocol is organized in cycles of equal length. These cycles are split in two parts, both of a particular length which remains equal for each cycle. The length of the regular part depends on the number of controllers in the network. There must be at least eight controllers in the network and at least the same number of slots in the regular part than controllers in the network. Each controller must be able to send at least once on each channel in the regular part. The extension's length can be any number of slots. The regular part is divided into two phases, the request and confirmation phase.

The following timed grammar shows the arrangement of the protocols in cycles, parts and phases.

$$
\begin{aligned}
cycle &\rightarrow \Box\,(cycleOfst \,.\, regPart \,.\, extPart) && (5.1)\\
regPart &\rightarrow reqPhase \,.\, confirmPhase && (5.2)\\
extPart &\rightarrow slot \,.\, \ldots \,.\, slot && (5.3)\\
reqPhase &\rightarrow slot \,.\, \ldots \,.\, slot && (5.4)\\
confirmPhase &\rightarrow slot \,.\, \ldots \,.\, slot && (5.5)\\
cycleOfst &\rightarrow \texttt{clkOfst} \,.\, \ldots \,.\, \texttt{clkOfst} && (5.6)
\end{aligned}
$$

where $|reqPhase| = |confirmPhase|$, since request and confirmation phase must have the same length. The cycle is introduced by a short amount of time $cycleOfst$, which is needed to overcome the maximum offset between the clocks of the controllers which could not be corrected by clock synchronization. Then, all controllers are in the first slot of the same cycle.

Production 5.1 shows the regular part followed by the extension part. Production 5.2 shows the two phases (request and confirmation). Both phases and the extension part consist of subsequent slots (production 5.3, 5.4 and 5.5). Note, that there can be empty slots in the extension. Since this has no influence on the length of the slots (which is static), these slots appear also as "regular" extension slots here.

Although, different controllers are active on the different channels in one slot, the grammar will not differ between sender and neighbor on channel **A** and sender and neighbor on channel **B**, because the behavior of the controllers is the same on channel **A** and channel **B**.

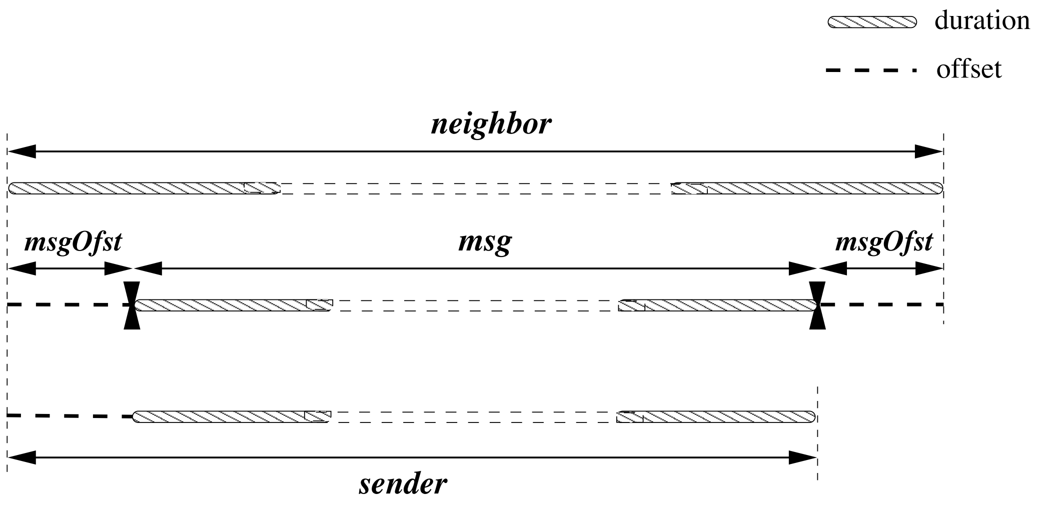


Figure 5.2: The duration of the *neighbor*–action is the time where the neighbor opens the gate. The offset $msgOfst$ is the time the sender waits before sending. The *sender* action is constructed by a sequence of waiting offset time $msgOfst$, and the time of sending sending the message (*msg*). The neighbor has no offset. It is starts immediately with opening of the gates.

When a controller enters a slot it must overcome the timing differences between the controllers' clocks. Therefore, each controller has to wait a small amount of clock units before the transmission phase can start. Then, it is assured that all controllers are in the same slot.

$$slot \rightarrow \square\,(slotOfst\,.\,transfer) \tag{5.7}$$

$$slotOfst \rightarrow \texttt{clkOfst}\,.\,\ldots\,.\,\texttt{clkOfst} \tag{5.8}$$

Within a slot, a controller can have one of three different roles. It can appear as sender, as neighbor or as receiver. After reception, the message transfer is done. All senders, neighbors and receivers must have finished their respective tasks. Therefore, the $transfer$ action includes all actions of the controllers, which run concurrently.

$$transfer \rightarrow sender \,||\, neighbor \,||\, receiver \tag{5.9}$$

Before the sender can start to send, the respective gate on that channel must be opened by the neighbor. Since the neighbor can be slower than the sender, it is safe to wait at least the same amount of time as during the transition from one slot to the next slot, to overcome the maximum possible clock differences between neighbor and sender. Therefore there is a message offset $msgOfst$, where the number of clock units is the same as the number of clock units of $slotOfst$. After waiting for the neighbor, the sender can send its message, which is a sequence of message header (*msgHdr*) and message body (*msgBdy*). The length of the message in clock units depends on the application requirements. Figure 5.2 shows the relationship between the actions *neighbor*, *sender*, $msgOfst$ and *msg*.

$$sender \rightarrow msgOfst\,.\,msg \tag{5.10}$$

$$msgOfst \rightarrow \texttt{clkOfst}\,.\,\ldots\,.\,\texttt{clkOfst} \tag{5.11}$$

$$msg \rightarrow msgHdr\,.\,msgBdy \tag{5.12}$$

$$msgHdr \rightarrow \texttt{clkUnit}\,.\,\ldots\,.\,\texttt{clkUnit} \tag{5.13}$$

$$msgBdy \rightarrow \texttt{clkUnit}\,.\,\ldots\,.\,\texttt{clkUnit} \tag{5.14}$$

and $|msgOfst| = |slotOfst|$.

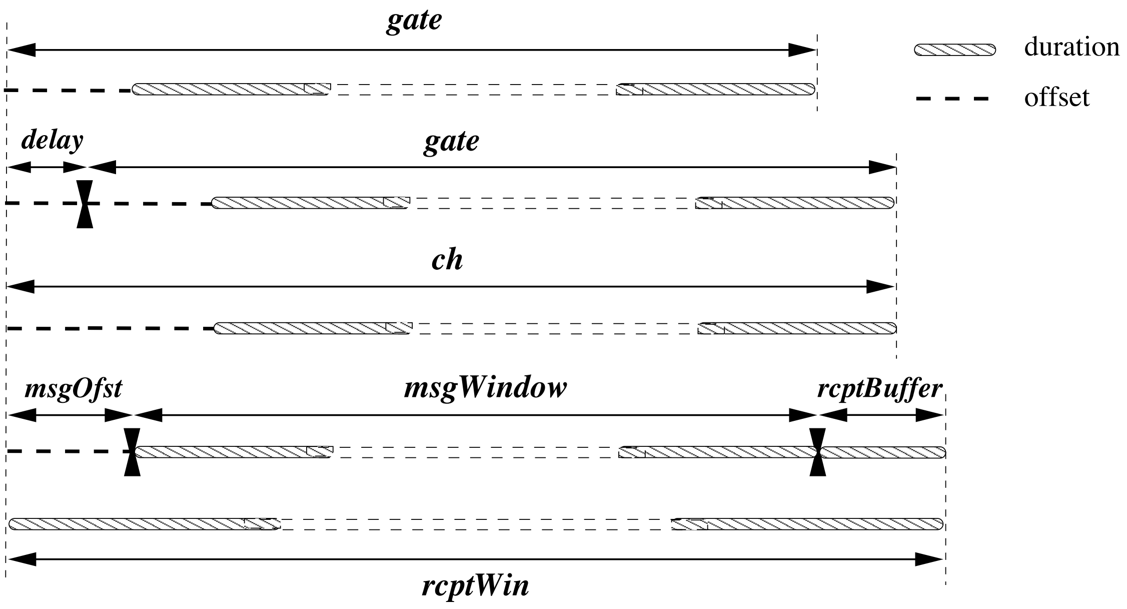


Figure 5.3: The *ch* action represents a delayed message, that passed the gate. The action *gate* at the top contains the original message (without delay). The delayed message must fit into the reception window.

The neighbor gives the sender the opportunity to send during the whole slot, as long as it can be sure that all controllers are in the same slot. This holds during the whole transmission phase. Therefore, the gate is opened before the first message offset starts. Then, the message window *msgWindow* follows, which is the time the sender needs to send its message. The neighbor closes the gate after another message offset to assure that a possibly very slow sender could send the whole message.

$$neighbor \rightarrow \square\,(msgOfst\,.\,msgWindow\,.\,msgOfst) \tag{5.15}$$

$$msgWindow \rightarrow msg \tag{5.16}$$

In the TEA protocol, the actions of a neighbor have an impact on the actions of a sender. Therefore, the neighbor controls a gate to allow or deny access to the channels. This means, that the sender's actions are limited by its neighbor.

$$gate \rightarrow sender \circ neighbor \tag{5.17}$$

Then, the message experiences a delay on the channel. The delays of the channels do not depend on the synchronized clock timing of the controllers, so that it cannot be given in terms of clock units. The action `delay` should include the maximum possible delay of both channels.

$$ch \rightarrow \texttt{delay}\,.\,gate \tag{5.18}$$

It is assumed that $\langle \texttt{delay} \rangle$ is an offset, which means $\Delta\,\langle \texttt{delay} \rangle = 0$.

The receiver waits for an incoming message by measuring the signal on the bus. The process of reception is completed if the message could be fully received. This means, that either the message ends, or the maximum message length is reached. An error occurs of the maximum message length is exceeded, and the reception must also stop. The maximum amount of time a message is accepted from the start of the slot is the *reception window*. It consists of the maximum message start delay, which is the message offset, the maximum possible channel delay and the maximum length of the message. The length of the reception window is measured in clock units, because this is the time base unit of every controller, so that the channel delay part of the reception window must be a multiple of clock units. Since the `delay` action describes the physical delay, an action must be available which is greater than the maximum physical delay and a multiple of clock units. This action is the *reception buffer*. Figure 5.3 shows the connection between the action *ch*, which represents an incoming message that recently passed the gate and being delayed by the channel, and the action *rcptWin*. This figure shows also the purpose of the reception buffer: As a consequence of the channel delay, the message is shifted into this buffer. The receiver must not stop receiving before the end of the buffer is reached.

$$\begin{aligned} rcv &\to ch \circ rcptWin && (5.19)\\ rcptWin &\to \Box\,(msgOfst\,.\,msgWindow\,.\,rcptBuffer) && (5.20)\\ rcptBuffer &\to \mathtt{clkUnit}\,.\,\dots\,.\,\mathtt{clkUnit} && (5.21) \end{aligned}$$

with $\min\Delta\,\langle rcptBuffer\rangle \geq \max\Phi\,\langle\mathtt{delay}\rangle$.

In the case of faults, it is possible that a message prematurely stops, so that it does not reach its desired length. In that case, a receiver could see a significant smaller slot. But this is not possible, since all slots must have the same size in TEA. Therefore, the reception window remains open, even if the received message is smaller than expected. Afterwards, the receiver waits for another message offset to assure that all other receivers closed the reception window within the same slot.

$$receiver \to (rcv \parallel rcptWin)\,.\,msgOfst \qquad (5.22)$$

The grammar does not contains multiple productions per slot, so that each partition has only one production. This means that there are no multiple views per entity, as long as all entities of terminal symbols contain only one view. This is the case for $\langle\langle clkUnit\rangle\rangle$ and $\langle\langle clkOfst\rangle\rangle$. Also, there are no multiple views for the delays on the channels and the offset at the start of the cycle.

5.4 Scheduling in TEA

The order of senders on the slots on both channels is determined by a schedule. There are two different schedules in TEA, each with different requirements.

The static schedule is used in the regular part. It is pre-configured and never changes during runtime. The function $\mathtt{sender} : \mathbb{N} \times \{\mathbf{A}, \mathbf{B}\} \mapsto C_t$ determines the sender within a given slot on a given channel (**A** or **B**), where C_t is the set of all senders.

Furthermore, each sender has a neighbor controller. This topic is explained in the next chapter. The function $\mathtt{neighbor} : C_t \mapsto C_t$ determines the neighbor of a controller. If controller j is the neighbor of sender i, then i is also the neighbor of j, so that $\mathtt{neighbor}(i) = j \Leftrightarrow \mathtt{neighbor}(j) = i$ for $i \neq j$. A controller cannot be the neighbor of itself, so that $\mathtt{neighbor}(i) \neq i$ for all $i \in C_t$.

The following requirements must be satisfied:

1. All senders must send at least once. The minimum length of the regular part is the number of possible senders.
2. The request and conformation phases have the same length. If m is the number of slots in the regular part, then both the request and confirmation phases consists of $\frac{m}{2}$ slots. This requirement is also part of the grammar, where $|reqPhase| = |confirmPhase|$ is required.

3. Each sender sends at least once in every phase. Furthermore, each controller sends on different channels in the request and confirmation phases. A controller can send on different channels in a phase, if it sends more than once in this phase.

The second kind of schedule is the dynamic schedule which is used in the extension part. The dynamic schedule is built during the regular part and changes from cycle to cycle. The only requirement is, that if a controller s is sending in the extension slot e, then it sends synchronously on both channels.

6 TEA Network Architecture

Real-time protocols in broadcast networks require protection against potential faults to prevent collisions on the transmission media. Real-time protocols cannot simply re-sent messages until the transfer is successful. Fault-free communication requires the ability of all fault-free controllers to complete a transmission in a fixed amount of time even in the case of faults. All kinds of components in the network can become faulty. These issues can be addressed by a redundant network architecture. If the transmission media (the channel) is in suspect to possibly become faulty, the network architecture requires two channels. Then, fault-free communication in the sense that a message can be transmitted without becoming corrupted requires the message to be sent on both channels. Although most protocols do not require fault-free communication in case of channel faults, the TEA architecture should enable this type of fault-free communication in any case. In other words, TEA does not offer a non-fault-tolerant operation mode in contrast to other protocols. Other measures against faults are necessary to protect channels from possible collisions. Independent components such as guardians can be used to restrict access to the channels. It has to be taken into account that these additional devices can also become faulty.

TEA tolerates single- and double-faults in the network. The protocol guarantees that each controller knows the current schedule. This is the topic of chapter 8. As long as a correct schedule is available, either pre-configured (regular part) or dynamically built up by an agreement algorithm (extension part), the architecture described here is able to prevent faulty components from disturbing the communication.

6.1 Weaknesses of Current Solutions

Fault-tolerant TDMA protocols similar to TEA use two different approaches for protecting the channels from access by faulty controllers. Guardian solutions are used with TTP/C and currently the proposed solution in connection with FlexRay, while double-controller approaches are used in a SAFEbus environment (see [TTA03], [Fle04] and [HD93]). Guardian solutions can either be central guardians on a star coupler as part of the channels itself, or additional devices in the nodes. All of these solutions have their advantages and disadvantages. Possible solutions are presented in section 2.5.3.

Central guardians allow a precise timing, because they have a protocol controller that can observe incoming messages and therefore do clock synchronization. On the other side, central guardians require star topologies. Busses, for example, cannot be supported. Guardians as part of the controllers' node support all possible topologies. Fully independent guardians would require an additional clock in the node and extending the guardians by clock synchronization and message receiver units. This makes this solution complex and expensive. As an alternative, they could share the same clock with the controller. This can cause the guardian concept to fail in case of clock faults.

The SAFEbus double-controller solution offers a by far higher degree of redundancy, which leads to a cost intensive solution, too. It uses two bus pairs and doubles (almost) all units on the node.

6.2 The TEA Architecture as Alternative

A TEA network consists of two channels **A** and **B**, and a set of an even number of controllers C_t (at least eight). Each controller is connected to both channels. TEA uses a double-controller approach instead of a guardian solution (see section 2.5.3). Each two controllers are grouped together into a single node. A sender controller is

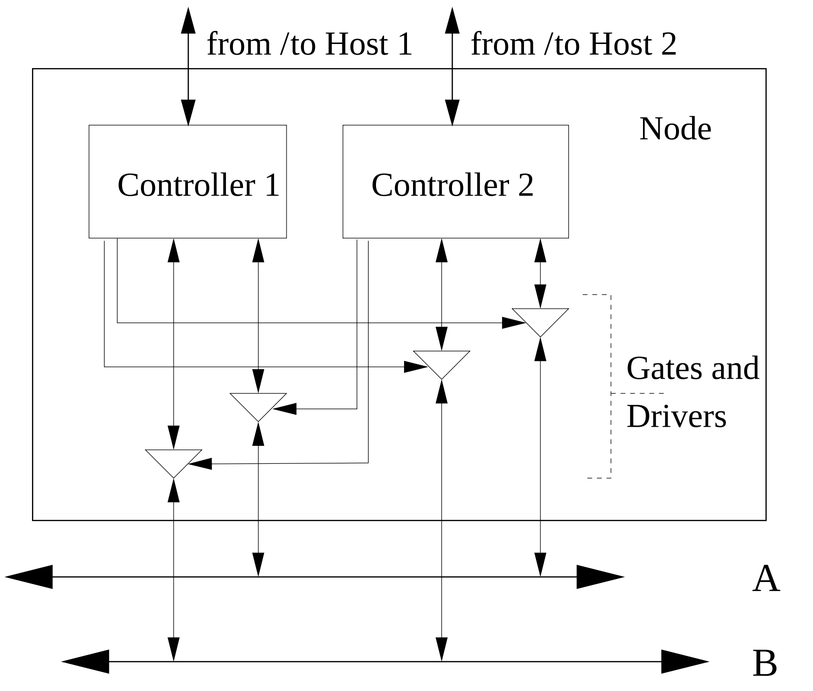


Figure 6.1: TEA nodes are using a double-controller approach where each controller guards its neighbour.

granted access to the channels by its neighbor controller which opens or closes a gate. They can serve different hosts. Both controllers are fully functionally independent units with independent clocks. The use of independent clocks assures that there is always one fault-free clock available.

Figure 6.1 shows the node architecture.

The following section 6.3 shows the network graph for a TEA network. It is composed of the sets of components (nodes in the graph) and the set of connections between components (edges in the graph). Every type of component is described in section 6.4, including the behavior in the fault-free and faulty case. Also, the component network behavior (see chapter 3) of each component is given. The subsequent section gives a runtime example for the fault-free case.

6.3 The TEA Component Network

A TEA network has four kinds of components: Controller, driver, gate and channel. Gates are composed of a driver and a switch. The following table lists the component types and the respective sets of the components of that type. The set of all components $\mathcal{K}$ of a network graph N is the union set $\mathcal{C}_t \cup \mathcal{D}_r \cup \mathcal{G}_t \cup \mathcal{C}_h$.

Component type	Set
Controller	$\mathcal{C}_t$
(Input) Driver	$\mathcal{D}_r$
Gate (Switch)	$\mathcal{S}_g$
Gate (Driver)	$\mathcal{D}_g$
Gate	$\mathcal{G}_t \subset \mathcal{D}_g \times \mathcal{S}_g$
Channel	$\mathcal{C}_h = \{\mathbf{A}, \mathbf{B}\}$

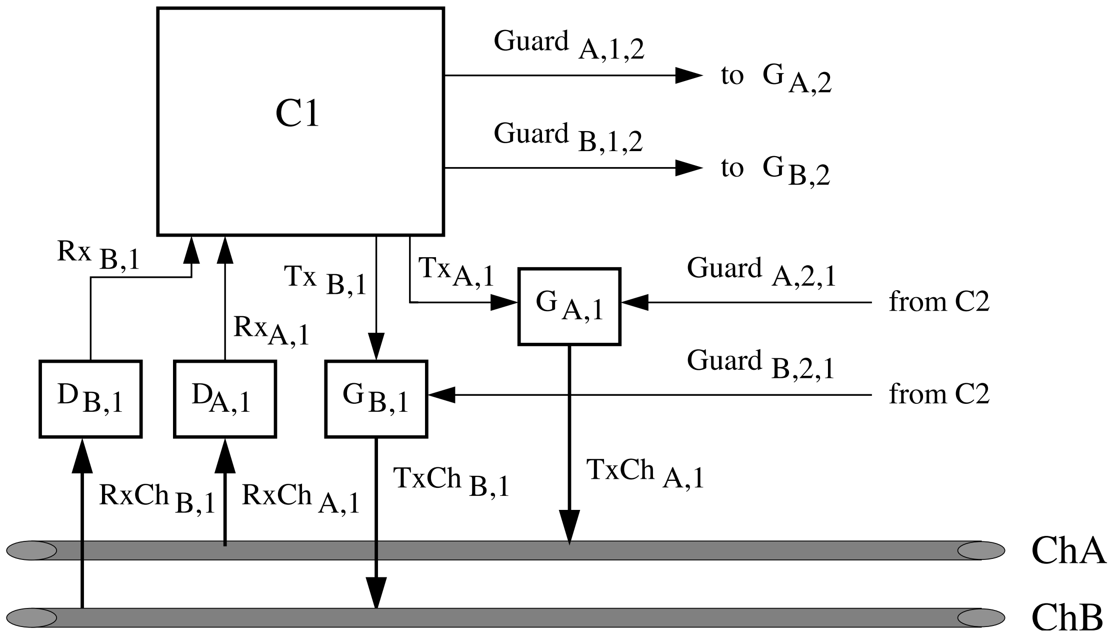


Figure 6.2: The controller C1, the driver ($D_{\mathbf{A},1}$, $D_{\mathbf{B},1}$) and gate ($G_{\mathbf{A},1}$, $G_{\mathbf{B},1}$) components and the connections between them.

Figure 6.2 shows the connection between the components. Note, that the channel is a component and not a connection.

	Connection	Set of connections	
1	$Guard_{\mathbf{c},j,i}$	*Guard*	Control connection from scheduler of controller j to gates of controller i on channel **c**
2	$Pwr_{\mathbf{c},i}$	*Pwr*	Connection between switch and output driver of gate of controller i on channel **c**
3	$Tx_{\mathbf{c},i}$	$Tx_{\mathbf{c}}$	Connection from controller i to driver on channel **c**
4	$Rx_{\mathbf{c},i}$	$Rx_{\mathbf{c}}$	Connection from driver on channel **c** to controller i
5	$TxCh_{\mathbf{c},i}$	$TxCh_{\mathbf{c}}$	Connection from driver of controller i to channel **c**
6	$RxCh_{\mathbf{c},i}$	$RxCh_{\mathbf{c}}$	Connection from channel **c** to driver of controller i

The following table clarifies the meaning of "activity" and "inactivity" at the respective connections in the fault-free case.

Connection	activity	inactivity
$Guard_{\mathbf{c},j,i}$	neighbor j sends a signal that turn the switch on	Switches are turned off
$Pwr_{\mathbf{c},i}$	the output driver is turned on	the output driver is turned off
$Tx_{\mathbf{c},i}$	a correctly coded message is sent to the gate	the connection is idle
$TxCh_{\mathbf{c},i}$	a correctly coded message passes the connection	the connection is idle
$RxCh_{\mathbf{c},i}$	a correctly coded message reaches the node from the channel	the connection is idle
$Rx_{\mathbf{c},i}$	a correctly coded message passes the connection	the connection is idle

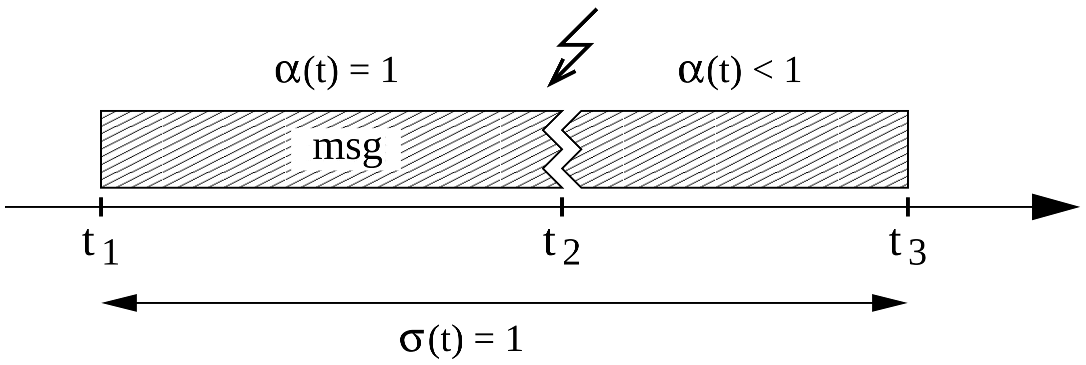


Figure 6.3: The activity at time t during message start and end, for example at $Tx_{\mathbf{c},i}$. The message starts at time t_1 and ends at t_3. The fault occurs at t_2. From that time on until t_3 the value $\alpha(t)$ becomes lower than 1.

In the following it is assumed, that the $Guard_{\mathbf{c},j,i}$ and $Pwr_{\mathbf{c},i}$ must be continuously held in an active state to keep the guard open. Alternatively, the switch could be triggered by an electrical impulse, and then keep its state until it is triggered again. The state of the other connections can either be idle, or the current signal is part of a correctly coded message. In particular there are three values $t_1 \le t_2 \le t_3$, such that the connection is active as long as the signal at time t lies between t_1 and t_2 and is part of a correctly coded message. The connection is regarded as active in this time interval. If, in case of a corrupted message, $t_2 \neq t_3$, then there is an encoding error at time t_2, while the (corrupted) message continues until $t = t_3$ (see figure 6.3). The connection shows weak activity or is inactive between t_2 and t_3.

The network is built up with nested compounds. Each controller, two output drivers and the switches which are part of the neighbors' gates are one compound. They exist twice on one node which is also a compound. These exist multiple times and are connected to the channels and the input drivers, which are not part of the node compound. The following table lists the compounds and their input and output interfaces.

Compound	Components/ Compounds	Interfaces input	Interfaces output	Connected to
$Ctrl_i$	$i \in C_t$ $d_{out,\mathbf{c}} \in \mathcal{D}_g$ $s_\mathbf{c} \in \mathcal{S}_g$	$Rx_{\mathbf{A},i}$ $Rx_{\mathbf{B},i}$ $Pwr_{\mathbf{A},i}$ $Pwr_{\mathbf{B},i}$	$TxCh_{\mathbf{A},i}$ $TxCh_{\mathbf{B},i}$ $Pwr_{\mathbf{A},j}$ $Pwr_{\mathbf{B},j}$	$Chan_\mathbf{A}$ $Chan_\mathbf{B}$ $Ctrl_j$
$Node_{i,j}$	$Ctrl_i$ $Ctrl_j$	$Rx_{\mathbf{A},i}$ $Rx_{\mathbf{B},i}$ $Rx_{\mathbf{A},j}$ $Rx_{\mathbf{B},j}$	$TxCh_{\mathbf{A},i}$ $TxCh_{\mathbf{B},i}$ $TxCh_{\mathbf{A},j}$ $TxCh_{\mathbf{B},j}$	$Chan_\mathbf{A}$ $Chan_\mathbf{B}$
$Chan_\mathbf{c}$	$\mathbf{c} \in C_h$ $d_{in,i,\mathbf{c}} \in \mathcal{D}_r$ for all $i \in C_t$	$TxCh_{\mathbf{c},i}$ for all $i \in C_t$	$Rx_{\mathbf{c},i}$ for all $i \in C_t$	all $Node_{i,j}$

If $i \in C_t$ is a controller then the controller j in the compound $Node_{i,j}$ is called the neighbor controller and vice versa. Any input driver physically located on a node, is not part of the node compound, but part of the connected channel.

6.4 Component Description

6.4.1 Channel

Messages are sent to other nodes via the two distinct channels **A** and **B**. Both share the same properties.

There are only few assumptions made about electrical properties. In the fault-free case, a channel propagates the signal from one input line to all out lines. The signal is not altered. The channels are shielded in a way, that one signal on a channel cannot influence the other channel. They also have an electrical signal delay in the range from $\delta^{\mathbf{ch}}_{min}$ to $\delta^{\mathbf{ch}}_{max}$.

Both channels are broadcast channels. TEA works with bus-, star- or mixed topologies, which are the most important in practice.

In case of faults on a channel, a message can become corrupted in three different ways.

1. The message gets lost, such that a receiver is not able to detect a transmission.
2. The physical encoding becomes corrupted.
3. The receiver can decode the physical encoding of the message, but some bits are corrupted in a way that the decoded value does not match the original message (bit flip, for example).

All three cases can be detected by a receiver. In the first case, a timeout can be used, because it is always known when the receiver has to be expect a message. The timeout must be greater than the maximum channel delay. Corrupted message content is detected by the use of a checkword (like CRC) mechanism (see section 2.2). It is assumed that the redundancy of the checkword is chosen high enough to recognize corruption in any case.

In case of message corruption, the channel can show byzantine behaviour. A subset of the controllers may receive non-corrupted messages, while the others receive either corrupted messages or no message at all.

Though a channel can contain active elements (star coupler for example), it is a passive component in the sense, that it cannot generate or duplicate messages. Also a receiver cannot misleadingly interpret channel noise as a correctly coded message, because the encoding (inclusive checkword) of a message is too complex.

The network behaviour of a channel **c** is

$$\phi(RxCh_{i,\mathbf{c}}, t) = \kappa(RxCh_{i,\mathbf{c}}, t) \prod_{j}^{|C_t|} \phi(TxCh_{j,\mathbf{c}}, t - d_{j,i})$$

with a constant $\delta^{\mathbf{ch}}_{min} \le d_{j,i} \le \delta^{\mathbf{ch}}_{max}$ which is the delay between sender j and receiver i.

Faulty behaviour is possible if there is activity on one of the inputs of the channel component, while inactivity is required and vice versa. In both cases, all outputs are affected. In case of channel faults, the effects at each output connection may be different. This means, that faulty behaviour of one of the inputs of the channels is not tolerated by the channel. Faulty behaviour on one of the inputs causes faulty behaviour on the outputs of the channel component, so that the receivers of a message are affected. In case of faults each individual connection to a receiver depends on the channel's component behaviour. This also means, that the individual outputs of a channel can be different in the case of faults (either channel fault or a fault within a node).

The component behavior function in the network behaviour function above assures that all cases of channel faults are included: If the channel remains silent though it should be active, messages get lost. The signals are not transmitted, and the channel is stuck in an idle state. On the other side, if the channel shows weak activity, the transmission becomes corrupted by channel noise, which prevents the signal to reach its desired state.

For the sake of completeness, it should be mentioned that there are scenarios possible, which are not covered by the above function. In the following, this scenarios will be ignored, because they are beyond the ability of fault-tolerant operation in TEA, anyway. For example, the following scenario shows a triple-fault: A faulty sender remains silent in its own slot, but another faulty controller, which can access the channel (maybe because one switch or the neighbor is faulty) sends a faked message, with header ID and similar information set to the original senders' values. No receiver is able to detect this kind of fault. It could be said, that the faulty behaviour of the three components is "hidden" by another fault in the value domain. But, the behaviour function model propagate an error condition to the receivers, because two inputs evaluate to a value lower than 1 in the product.

Although, this is beyond the scope of TEA, a more detailed channel model may take similar scenarios into account.

6.4.2 Input Driver

A broadcast massage passes the input driver before it arrives at the controller. A driver causes a delay between $drDelay_{min}$ and $drDelay_{max}$. Because a short-circuit should not have an impact on the channel an undesired reverse reaction is made impossible (by an additional resistor, for example).

A faulty driver may deface a message in a similar way as a channel does. The physical encoding or the content may be affected.

The network behaviour of a driver $d \in \mathcal{D}_r$ connected to channel $\mathbf{c}$ and controller i is

$$\phi(Rx_{\mathbf{c},i}, t + d_{\mathbf{c},i}) = \kappa(Rx_{\mathbf{c},i}, t + d_{\mathbf{c},i})\, \phi(RxCh_{\mathbf{c},i}, t)$$

with $drDelay_{min} \leq d_{\mathbf{c},i} \leq drDelay_{max}$.

A faulty driver can only have an impact on one connected controller with respect to the input from a single channel.

6.4.3 Controller

A controller is responsible for sending messages at the right time within a slot. Additionally, it must grant its neighbor controller access to a channel, if the neighbor is scheduled for that channel in that slot. Otherwise, access to the channels must be denied. In any slot it must be able to receive messages from the channels.

Although a controller is one single component, it has three distinct functions (gate control, sending and receiving) which can be analyzed separately. This is possible, because there is no direct relation between their functionality at the same time. This means, for example, that received messages can only affect the behavior of a controller in subsequent time steps.

Another part of the controller is the scheduler unit. It is responsible for assigning a role to the controller (sender, neighbor or receiver only) depending on the schedule and the current slot. Therefore, it has an impact on the sender and gate control, while the receiver is always active, because a controller acts always as a receiver. Figure 6.4 shows how the different functional units are related to each other.

The communication controllers need a global time base provided by the clock, which is regarded as the corrected time which is derived from the oscillator frequency. This means the oscillator and the controller are regarded as one unit, because clock synchronization lies in the responsibility of the controller. TEA makes no assumption about the underlying clock synchronization algorithm, as long as a synchronized global time base can be provided to all fault-free nodes.

As shown in section 5.3 the slot is divided into different phases, depending on the role the controller takes in the current slot. Any controller starts with the slot offset $slotOfst$, which appears at the beginning of the slot. If the controller is the current sender or neighbor, then the message offset $msgOfst$ (second and fourth phase) and the message window $msgWindow$ follows. In any case the controller is also a receiver, so that the reception window $rcptWin$ runs in parallel within the slot. The slot ends, if the four phases of sender and neighbor, as well as the reception window are completed. In the following, the purposes of each of these phases are explained in detail.

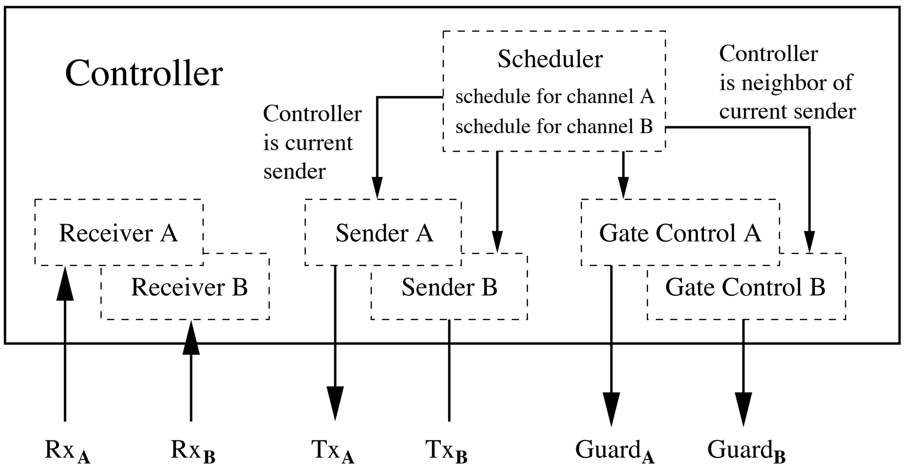


Figure 6.4: The logical division of a controller in receiver, sender, gate control and scheduler. The controller is able to receive at any time, while the sender and gate control parts are activated depending on the controllers' role in the schedule.

Section 6.5 shows the timing of the different phases in a slot within a simulated hardware environment (sender, scheduler and gate control).

The controller is a possible source for a wide variety of failures. In general, a faulty controller can behave arbitrarily at any time. Faulty behaviour can occur independently at all interfaces of the controller.

Scheduling

The problem of scheduling is related to slot accounting, and determining the current sender, opening the neighbors gate if necessary and triggering the own sending process if necessary. The term "scheduling" is used in two slightly different meanings in connection with the TEA protocol. The *static schedule* is the fixed assignment of a controller's sending rights to a channel in a slot in the regular part. This static scheduling is discussed in this section. *Dynamic scheduling* deals with the order of senders in the extension, which can be different in every cycle depending on the requests of the controllers and the current policy. An advanced discussion about dynamic scheduling and related topics are subject to chapter 8.

Each controller has a static schedule register `sched` which is set at configuration time (before runtime). This register contains the fixed schedule for the regular part only. It is different for channels **A** and **B**. Alternatively, it may be possible to calculate the fixed schedule on the fly from the identification numbers of the controllers. The schedule register contains the identification number of the current sender in slot s on channel $\mathbf{c}$ in cell $\texttt{sched}_{s,\mathbf{c}}$.

The controller has to send a message if $\texttt{sender}(s,\mathbf{A})$ or $\texttt{sender}(s,\mathbf{B})$ is the controller itself in the current slot s. Otherwise, it expects a message from the currently scheduled senders on the respective channels. If the current sender on one of the channels is the neighbor controller, it has to open the gate for the length of the slot.

Similarly, there is also a dynamic schedule register `dyna_sched`. This is a bit vector, where each controller corresponds to one bit position. A bit is set if a controller requested a slot. A controller determines the next sender using a scheduling policy, for example a simple round-robin scheme. In this case, there is an index register which points into the `dyna_sched` register. If a slot in the extension part starts, the controller corresponding to this position is the next sender. The bit is then cleared, and the index is moved forward to the next position with a set bit. If the index reaches its maximum value, it is wrapped around and starts from the first entry in the `dyna_sched` register. Alternative methods for dynamic scheduling are presented in chapter 10.

The slot accounting depends on the controller's clock. TEA uses a distributed time base, which must be synchronized in regular intervals. A maximum inaccuracy is taken into account which leads to a small time shift between the fastest and slowest controller. The maximum difference between the current time of these controllers is δ^c_{max}. This value is the basis of a *clock unit*, which is the time unit of the synchronized time base in TEA as presented in section 2.5.2.

A faulty controller may send in the wrong slot. If the clock is faulty and the clock rate exceeds the specified maximum tolerated drift, then the controller may send before the gate is opened or after it has been closed. If the controller is the neighbor of the current sender it may open the gate too recent, or too late. The gate may also be closed too early or too late. In general it is assumed that a faulty controller starts sending at any time and for an arbitrary amount of time. It can open or close the gate at any time, also.

Gate Control

A controller i has to control its neighbors gate to allow or deny the neighbor to access the channel $\mathbf{c}$. If the neighbor of i is not scheduled for sending, then the connection $Guard_{\mathbf{c},i,j}$ from controller i to the gate of neighbor j on channel $\mathbf{c}$ must be inactive. Otherwise, the controller has to allow its neighbor to send on the channel in the current slot.

To assure that the sender controller is in the same slot as the neighbor controller, the neighbor has to wait $clkDiff_{max} = \delta^c_{max}$ (see section 2.5.2) before the gate can be opened. Section 6.5 shows the process of guarding and sending at runtime. This is the slot offset (phase one) with $\Phi\ \langle slotOfst\rangle = clkDiff_{max}$. During the phases two to four, connection $Guard_{\mathbf{c},i,j}$ must be active (action $msgWindow$).

In case of a faulty controller, the gate can be opened and closed at any time. The behaviour function for the gate control is

$$\phi(Guard_{\mathbf{c},i,j}, t) = \kappa(Guard_{\mathbf{c},i,j}, t)$$

for controller i and sender j in the same node.

Since the controller is regarded as one single unit, the schedulers influence on the network behavior is already part of the component behavior.

Receiving

Even if a controller is the current sender or neighbor in the slot it acts as receiver at any time. A receiver accepts a message after the slot offset until the end of the reception window.

The reception window consists of the message buffer, the message window and the reception buffer. The purpose of the reception buffer is to balance out the channel delay. For this reason, the length of the reception buffer in clock units must be higher than the maximum channel delay (which is given in a physical unit, for example *ns*). The whole length of the reception window must be sufficient to contain the maximum length of a message (which can be greater than expected, because the sender can have a slower clock than the receiver) and the maximum delay.

When a message start is detected, the controller starts with the physical decoding of the signals on the channels. The process of reception can be aborted on a decoding error. It ends in any case, after an idle channel state is detected or the end of the reception window is reached. A channel is idle, if there is no activity on the channel until a timeout passes. If the reception has been stopped prematurely, message corruption is signaled.

If the message has been decoded successfully, the checkword is verified. If this fails, the receiver reports a corrupted message, else the protocol-relevant information in the message is read out, and the payload is sent to the host. In case of message corruption, the message is ignored including its partially decoded information.

Depending on the message format, it is possible to verify parts of the message, while subsequent parts of the message are not fully decoded. This may be important if parts of the message are needed as soon as available, for example protocol-relevant header information. In this case, the message can have two checkwords, one header checkword which is located near the start of the message, and a payload checkword located near the end of the message.

Under special circumstances, one aspect of timing can become important. The decoding process consumes a small amount of time. This is true for physical decoding, but especially for the verification of parts of a message during reception. If there is protocol-relevant information encoded in the message which must be extracted during reception, the signal decoding continues. Since the controller must wait until the information is fully decoded, there is an offset between the arrival of a signal and the time a receiver can react. This offset is called the *decoding delay*. In TEA, the decoding delay becomes important in connection with the possibility to send messages of dynamic length in the extension part. This will be discussed in chapter 11.

After the reception controller i has decided if a message was received correctly or not. The result of the reception process is $\mathrm{RES}(\mathbf{c}, i)$. The result can be different on each channel. If the sender sends synchronously on both channels, only one message must be received correctly on one channel, for successful reception of a message. This is the case in the extension part. The reception process starts with an incoming message $rcvMsgStart_{\mathbf{c},i}$ and ends with correct and complete message processing or abortion at $rcvMsgEnd_{\mathbf{c},i}$. The result function is

$$\mathrm{RES}(\mathbf{c}, i) = \prod_{t=rcvMsgStart_{\mathbf{c},i}}^{rcvMsgEnd_{\mathbf{c},i}} \theta(Rx_{\mathbf{c},i}, t-d, a_0(Rx_{\mathbf{c},i}), a_1(Rx_{\mathbf{c},i}))$$

where $t - d$ is the time before the decoding (this means d is the decoding delay) and $a_0(Rx_{\mathbf{c},i})$ and $a_1(Rx_{\mathbf{c},i})$ are the tolerances of controller i. The result is regarded as faulty, if the value $\theta(Rx_{\mathbf{c},i}, t-d, a_0(Rx_{\mathbf{c},i}), a_1(Rx_{\mathbf{c},i})) = 0$ during $rcvMsgStart_{\mathbf{c},i} \leq t \leq rcvMsgEnd_{\mathbf{c},i}$. This means, the controller does not tolerate faulty behaviour. It is assumed, that the values of $a_0(Rx_{\mathbf{c},i})$ and $a_1(Rx_{\mathbf{c},i})$ are the same for all controllers, and that $a_0(Rx_{\mathbf{A},i}) = a_0(Rx_{\mathbf{B},i})$ and similarly $a_1(Rx_{\mathbf{A},i}) = a_1(Rx_{\mathbf{B},i})$ for simplicity.

Although a sending controller can send wrong content in the message, it should be considered as fault-free behaviour if the message is correctly encoded (including correct checkword). Wrong payload data content must be handled by higher network layers. Wrong values for protocol-relevant information can either be detectable (a sender identification field in the message, for example) or must be handled by other mechanisms. Chapter 8 shows how wrong request bits and confirmation vectors are tolerated.

Sending

A controller acts as a sender if it has sending permissions for a particular slot on a particular channel. In TEA, the permissions are granted if the controller is assigned for this slot on this channel (regular part) or it was chosen for a slot in the extension part. In the first case, the assignment to a slot and channel is pre-configured before runtime. In the extension part the slots are chosen dynamically based on agreement.

When sending, the controller is responsible for encoding a message and sending the resulting signal to one (regular part) or both (extension part) channels. Before the signal actually reaches the channel it has to pass the output driver.

The sender encodes messages in two steps. Payload data is assembled together with header information, possibly a request bit or confirmation vector for the allocation of slots and other protocol related data (see chapter 8). The messages are protected by a checkword, like CRC, or cryptographic signatures. Afterwards, the message is physically encoded for transmission over the channel.

With the start of the message offset, the neighbor grants the sender access to the channel by opening the respective gate at the beginning of the first message offset and keeping it open until the end of the second message offset. After this point, the sender has no more access to the channel. The sender sends the encoded message in the message window, which is the third phase in the slot. The message offset assures, that the controller can be sure the gate is open. This is necessary, because the neighbor can be slower than the sender since there are small time variances in the clocks of every controller. The message offset prevents the sender to send while the gate is closed. Similarly, the sender must stop sending a small amount of time before the neighbor closes its gate. Clock variances may cause the neighbor to be faster than the sender. Chapter 9 analyzes the details about timing.

The message has always constant length. The message fills almost the whole message window. A fault-free controller must send a message, even if there is no payload to send. In that case, the message should be marked as null message.

Since the controller has only access to the channel if the neighbor allows it, it is not possible that a faulty controller disturbs ongoing transmissions from fault-free controllers, given that gate and neighbor are fault-free. Within its slot a controller can send any kind of signal. A random signal leads to decoding errors at the receiver and is rejected there.

Sender faults can have an impact on a single or both channels. For the network behaviour function only the time of sending is important. A sender can send at the wrong time or it may not send although it should. The behaviour function is

$$\phi(Tx_{\mathbf{c},i}, t) = \kappa(Tx_{\mathbf{c},i}, t)$$

for controller i at channel **c**. The sender does not depend on the inputs of the controller.

6.4.4 Gate

To assure that a controller does not cause collisions by accessing a channel at the wrong time, the neighbor controller can avoid any activity by closing a gate on the message transfer line from the controller to the channel. On the other hand, the neighbor should open the gate as long as the controller is allowed to send.

Gates are divided into two subcomponents. A switch can disconnect the output driver from power. The switch is directly controlled by the neighbor controller of the sender by opening and closing the gate. Figure 6.5 shows this arrangement.

The output driver shares the same properties as the input driver. An electrical signal experiences also a delay in the same interval. The difference to the input driver is the additional connection $Pwr_{\mathbf{c},i}$. Unless this connection is active, physical access to the channels is impossible. The behavior of a faulty controller is not forwarded to the output connection of the gate. If $Pwr_{\mathbf{c},i}$ is active, the sender has physical access to the channel, and (faulty or fault-free) behavior at the channel depends on the behaviour of the sender. This requires access rights of the sender in the current slot.

On the other side, if the behaviour at $Pwr_{\mathbf{c},i}$ is faulty, then the result at $TxCh_{\mathbf{c},i}$ is fault-free, if the controller connected at $Tx_{\mathbf{c},i}$ does not send currently. Otherwise, the resulting behavior at the connection $TxCh_{\mathbf{c},i}$ depends on the behaviour at $Pwr_{\mathbf{c},i}$.

If a sender has access rights, the protocol requires activity on connection $Pwr_{\mathbf{c},i}$, which means that $\sigma(Pwr_{\mathbf{c},i}) = 1$. During this time both controllers must show fault-free behavior. This is the case during the *msgWindow* action of the neighbor.

The behavior function of the gate is split into three parts. The first part represents the output driver without being affected by the switch. This function is similar to the input drivers' network behavior.

$$\phi^1(TxCh_{\mathbf{c},i}, t) = \kappa(TxCh_{\mathbf{c},i}, t)\,\phi(Tx_{\mathbf{c},i}, t')$$

with $t' = t - d_{\mathbf{c},i}$ and $drDelay_{min} \leq d_{\mathbf{c},i} \leq drDelay_{max}$.

Similarly, the behavior of the switch depends on the connected neighbor of the controller the gate belongs to. In contrast to the driver, it is assumed that there is no delay.

$$\phi(Pwr_{\mathbf{c},i}, t) = \kappa(Pwr_{\mathbf{c},i}, t)\,\phi(Guard_{\mathbf{c},j,i}, t)$$

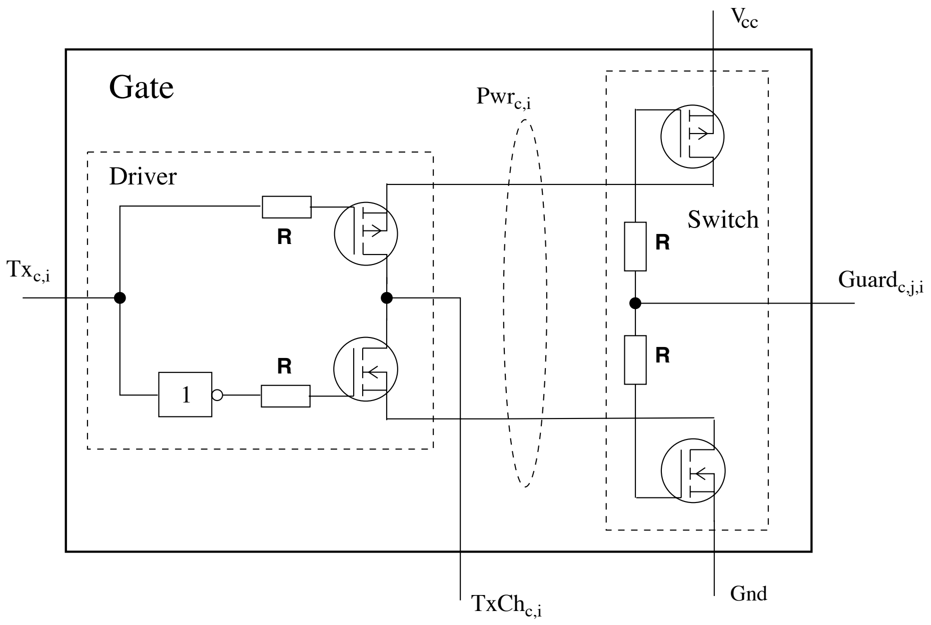


Figure 6.5: The gate is divided into the subcomponents output driver and switch. A neighbor controller can deny a sender access to the channel by de-powering the output driver. The resistors prevent the controllers from electrical damage in case of faulty components in the switch or the output driver.

The switch has the ability to disable the output driver, which means, the behavior of the switch is also part of the behavior of the gate. The behavior at connection $TxCh_{\mathbf{c},i}$ is the behavior of the driver, if it is enabled. Otherwise, the behavior of the gate may be faulty, if the protocol requires activity at the output connection. To decide, if the gate enables or disables the driver in case of faulty behavior, the network behavior function uses the protocol function $\sigma(Pwr_{\mathbf{c},i}, t)$.

If activity is required at connection $Pwr_{\mathbf{c},i}$, then the behavior depends on the behavior of output driver and switch. This also means, that the behavior becomes faulty, if the switch behaves faulty because it disabled the driver.

$$\phi^2(TxCh_{\mathbf{c},i}, t) = \sigma(Pwr_{\mathbf{c},i}, t)\, \phi^1(TxCh_{\mathbf{c},i}, t)\, \phi(Pwr_{\mathbf{c},i}, t)$$

If the driver should be disabled, faulty behavior means that the switch enabled the driver. Then, the behavior of the gate depends on the behavior of the driver.

$$\phi^3(TxCh_{\mathbf{c},i}, t) = (1 - \sigma(Pwr_{\mathbf{c},i}, t))\, (\phi(Pwr_{\mathbf{c},i}, t) + (1 - \phi(Pwr_{\mathbf{c},i}, t))\, \phi^1(TxCh_{\mathbf{c},i}, t)$$

The function shows, that weak activity can lead to faulty behavior depending on the behavior of the driver and the switch. If the driver behaves fault-free, then the resulting behavior is also fault-free regardless of the behavior of the switch. The reason is, that enabling or disabling the driver while inactivity is required has no effect on the behavior of the gate as long as the driver behaves correctly.

The network behavior of the gate is

$$\phi(TxCh_{\mathbf{c},i}, t) = \phi^2(TxCh_{\mathbf{c},i}, t) + \phi^3(TxCh_{\mathbf{c},i}, t)$$

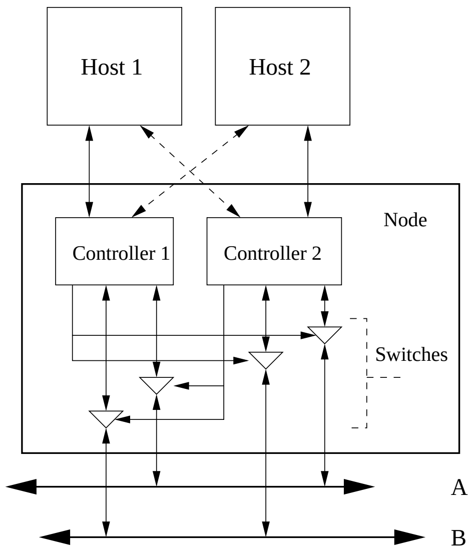


Figure 6.6: One possible way to use both controllers: Multiple hosts can be connected to one controller each or each host can be connected to both controllers (denoted by the dashed lines).

6.4.5 Host Interconnection

The controllers can be connected to the hosts in different ways. In the regular case, there is one host connected to one controller. But it is also possible to use both controllers for a single host in order to improve redundancy. Since there are two controllers on one node, each controller can serve a host of its own. Also, it is possible that two or more hosts are using the interface of both controllers as illustrated in figure 6.6.

In the context of this work, considerations about possible faults in the host interface are not of interest. These kind of faults must be handled on higher protocol levels.

6.5 Runtime-Example

In the following, an example Verilog implementation for the parts of a TEA controller discussed above will be given for illustration. The results of an example simulation run will also be shown. This implementation concentrates on the operation within a slot. Further topics like dynamic scheduling and clock synchronization are not part of the implementation. On the other side, the slot control mechanism could be synthesized for an FPGA.

The implementation of each controller consists of a sender unit, a gate control unit, the sender and a scheduler unit. This example uses an extra central control unit to keep track of the phases and trigger the other units to become active. The test bench simulates one slot, which is shared by two instances of a controller. Therefore, the schedulers are configured for one slot only. Both controllers are their respective neighbors.

Also, there are four instances of a gate module. The outputs to and from the channels are connected by a channel module. Its purpose is to simulate a channel delay between the two controllers. The test bench also generates two clock signals, one for each controller.

Below, the core part of the control unit is shown. It is triggered at every clock tick. The time between two clock ticks has been set to $10\,ns$. The code uses some macros (`start_of_message_offset_1`, ...), which contain the number of clock ticks which must pass within the current slot until the respective phase starts.

The inputs `neighbor_on_A`, `neighbor_on_B` and `sender_on_A` and `sender_on_B` are connected to the scheduler of the controller, which provides the information about the current role within the schedule.

```

// Controller external input from clock.
// A positive edge is generated each 10 ns
// It triggers the' always'-procedure below
input clock;
wire clock;

// Controller external input to gates of the neighbor controller
// 1-bit register output. If set to 1, the neighbors' switch opens
// the output driver on the respective channel A or B
// initially, the registers are set to 0
output guard_A, guard_B;
reg guard_A, guard_B;

// Controller internal connection from scheduler
// 1-bit, is set if controller is neighbor or sender on channel A and/or B
input neighbor_on_A, neighbor_on_B, sender_on_A, sender_on_B;
wire neighbor_on_A, neighbor_on_B, sender_on_A, sender_on_B;

// Controller internal connection to receiver and sender
// if reception window is set to 1, then an incoming message is expected
// send message to respective channel, if send_message_A/B is set to 1
output reception_window, send_message_A, send_message_B;
reg reception_window, send_message_A, send_message_B;

// internal register
reg[2:0] phase; // phase counter
reg[19:0] slot_time; // number of clock ticks in the current slot

always @ (posedge clock) begin
  slot_time = (slot_time + 1) % `slot_length;
    // 'phase' is set to 1 at the beginning of the slot (slot offset)
    // When the message offset starts, the gate on the
    // respective channel must be opened (regular part)
    // In the extension part, both gates must be opened
    // This information is send by the scheduler part
  if (slot_time >= `start_of_message_offset_1 && phase == 1)
  begin
    phase <= 2;
    if (neighbor_on_A)
      guard_A <= 1;
    if (neighbor_on_B)
      guard_B <= 1;
  end
    // start sending if sender on the respective channel
  if (slot_time >= `start_message_window && phase == 2)
  begin
    phase <= 3;
      // signal receiver that a message is expected
    reception_window <= 1;
    if (sender_on_A)
      send_message_A <= 1;
    if (sender_on_B)
      send_message_B <= 1;
  end
    // still receive messages, but stop sending
  if (slot_time >= ‘start_message_offset_2 && phase == 3)
  begin
    phase <= 4;
    send_message_A <= 0;
    send_message_B <= 0;
  end
    // close the gates 1 tick before the message offset ends
    // neighbor is not allowed to send anymore
    // incoming signals furthermore possible, depending on
    // channel delay
  if (slot_time >= ‘end_message_offset && phase == 4)
  begin
    phase <= 1;
    guard_A <= 0;
    guard_B <= 0;
  end
    // do not expect any more incoming signals
  if (slot_time >= ‘end_reception_window)
  begin
    reception_window <= 0;
  end
  // new slot starts, if slot_counter == slot_length this is
  // assured through the modulo at the start of the procedure
end
```

Figure 6.7 shows the simulated slot.

The waveforms at the connections of the first controller are shown in the first half, the connections of its neighbor in the lower part in the same order. The first entry shows the Id number of the respective controller (a 6-bit register in this example). The values used are 01 for the first, and 02 for the second controller. They are pre-configured

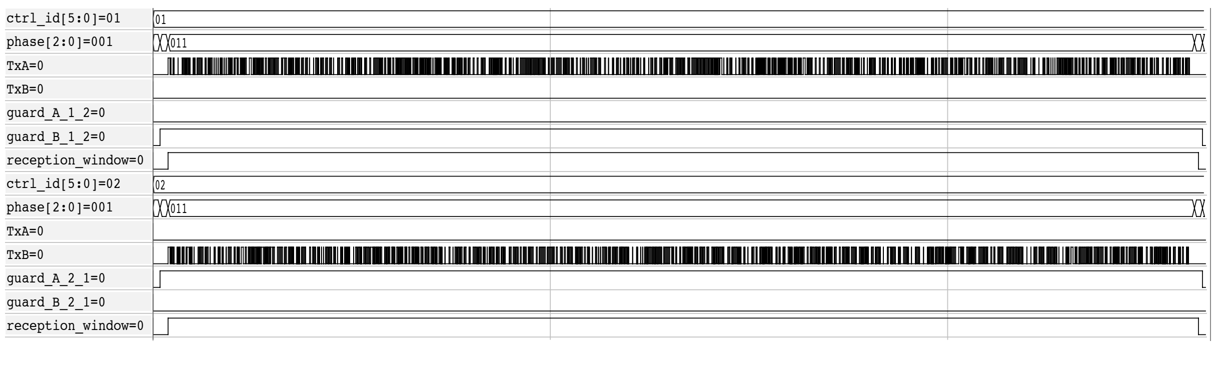


Figure 6.7: A slot in the regular part of a cycle in an example run. The phase signal follows the operation of the sender and neighbor controllers.

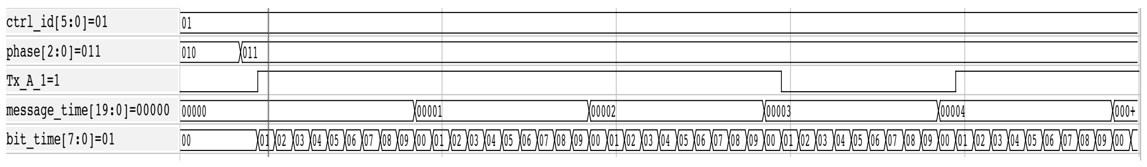


Figure 6.8: The message of controller 1 starts with a three bit sequence at HIGH level, the first byte follows with a LOW-HIGH-bit sequence. In this example, it has to be taken into account that the sender reacts one bit-time tick later before adjusting the signal-level on Tx.

and never change during runtime. The second signal shows the phase counter, which is a 3-bit register (for four phases). Note, that this is given as binary number. The phase counter is used by the current sender and neighbor only. The receiver units are controlled by a *reception_window* register, which is a single-bit register which is set if the receivers should be ready for an incoming transmission and cleared, if no more signals should be accepted. The next two signals are the TxA and TxB outputs. Since controller 1 is scheduled as sender for channel **A**, it sends a signal in phase three and remains silent on the other channel. Its neighbor on the other side is sender on channel **B**. This also means that the respective other controller has to open the gate of the neighbor's channel. The following two signals show the state of the gate control signals for both channels ($guard_{\mathbf{c},i,j}$ for $i, j \in \{0, 1\}$ and $\mathbf{c} \in \{\mathbf{A}, \mathbf{B}\}$). The next signal shows the controller's internal reception window signal. It shows HIGH-level if the controller is ready to accept a message from the channels. The last two signals show the resulting delayed signals from the channels. Below, there are the same informations for controller 02.

The sender unit starts sending if the control unit sets the `send_message_A` or `send_message_B` to 1. Both ports are checked with every clock tick. The sender sends the message to one or both channels.

```
module Sender (clock, TxA, TxB,
        send_message_A, send_message_B);

input clock;
// controller internal input from control unit (see above)
input send_message_A, send_message_B;
output TxA, TxB; // controller external outputs to output drivers

wire clock;
wire send_message_A, send_message_B;
reg TxA, TxB;
```

The implementation uses an example physical encoding, where the message is split into three parts. The first part is the start sequence of the message. It is a sequence of 3 bits set to HIGH-level. Figure 6.8 shows the start sequence of the message from the first controller. Below the message bit counter (message_time) of the controller is shown, a 20-bit counter which counts the bits from the start of the message.

The second part is the message itself. Each byte is encoded in 10 bits. They start with a LOW-HIGH-sequence, to mark the start of a new byte. Then eight data bits follow (random bits in this implementation). The message ends if there are 12 consecutive bits set to LOW-level. This marks the idle state of the channel. Messages have a maximum length. If a receiver cannot detect an idle state it stops after the maximum number of bits are decoded. This is regarded as decoding error.

The function `signal_level` determines the value of the current output signal.

```
function signal_level;
input [7:0] cur_bit_time;
input [19:0] bit_counter;
input data_bit;
begin
    // test part of the (physical) message encoding
  if (bit_counter < ‘message_start_seq)
       // frame start is always 1 (first three bits)
    signal_level = 1;
  else if (bit_counter < ‘message_start_seq + ‘message_body_len)
  begin
      // this is the message body
      // bytes are encoded in 10 bits
      // the first two of them are always set to ’01’
    if ((bit_counter - ‘message_start_seq) % ‘byte_length == 0)
        // first bit
      signal_level = 0;
    else if ((bit_counter - ‘message_start_seq)
           % ‘byte_length == 1)
        // second bit
      signal_level = 1;
    else
      signal_level = data_bit;
    end
  else  // last 12 bits are set to ’0’
    signal_level = 0;
  end
endfunction
```

In this function `bit_counter` is the number of the current bit in the signal. It is incremented every time a bit-time counter reaches its maximum value. The counter `bit_time` is increased with every clock tick. The maximum value is 10. Therefore, with 10 ns per clock tick, the bit length is 100 ns. The following definitions are part of the sender module.

```
reg[7:0] bit_time; // in clock ticks
reg[19:0] message_time; // counts the bits in the current message
```

The implementation includes an example for physical encoding only. Higher level encodings with checkword generation and assembling protocol-relevant data are excluded.

7 Behavior in the Presence of Faults

The purpose of the TEA architecture is to assure collision free communication. Fault-free controllers should be able to send at least once on at least one channel per cycle. This chapter shows, what kind of faults are possible, what impact faulty components have on the operation of the protocol and if they are tolerated.

The protocol is intended to tolerate single- and double-faults. In TEA, there is a difference between component and compound faults. In a component network, single components can behave faulty on at least one of their outgoing connections. But all components are part of a channel or controller compound, too. These were defined in section 6.3. In contrast to a single component fault, a single compound fault is present, if there are one or more faulty components in one controller or channel compound. If a compound contains a faulty component, then the compound is said to be faulty. A double-fault is present, if one controller and one channel compound are faulty.

Sections 7.1 and 7.2 focus on the effect of single component faults on transmissions. The remaining chapters deal with compound faults.

Faulty behavior is present in a component network, if $\kappa(v, t) < 1$ for at least one connection $v \in \mathcal{V}$ and $t_{start} \leq t \leq t_{stop}$.

Faulty behavior is tolerated, if the transmissions are successful even in the case of faulty behavior. Two requirements must be fulfilled:

- Faulty components do not corrupt the transmission of a current sender, if the sender is not part of the same node as the faulty component. The node architecture is responsible for this property. This is the topic analyzed in this chapter.
- Every controller is able to build a schedule for the extension part. All fault-free controllers must agree on the same schedule. This can be assured using the agreement algorithm presented in chapter 8.

The question if the transmission of a message is successful or not is discussed in the next section. This leads to the main topic of how a fault can affect an ongoing message transmission, and the possible different conclusions at the receivers.

7.1 Transmission Failures in TEA

This section contains some important definitions. Based on the network behavior functions of the components in a TEA network, as defined in section 6.4, the term *transmission failure* is introduced. Basically, a transmission failure occurs, if at least one receiver does not accept an incoming message. The following paragraphs will give a precise definition. Furthermore it will be possible to define the meaning of byzantine faults in connection with transmission faults.

A message has been transfered without faults in a slot $1 \leq s \leq |reqPhase| + |confirmPhase| + |extPart|$ on channel **c**, if

$$\prod_{i \in C_t} \mathrm{RES}(\mathbf{c}, i, s) = 1$$

where $\mathrm{RES}(\mathbf{c}, i, s)$ is the resulting behavior at $Rx_{\mathbf{c},i}$ during the reception window of slots s as shown below. In other words, a transmission is successful, if there is no faulty behavior during the reception window at all receivers. This is also the case, if there are faulty receivers, because the behavior of the receiver is not part of a receivers' result function in a TEA network.

On the other side, faulty behavior might be tolerated though inexact measurement. This is expressed using the thresholds $a_0(Rx_{\mathbf{c},i})$ and $a_1(Rx_{\mathbf{c},i})$ in the result function (see also figure 3.6). The result function in the reception window of slot s is

$$\mathrm{RES}(\mathbf{c}, i, s) = \prod_{t=listen_{start}(s)}^{listen_{end}(s)} \theta(Rx_{\mathbf{c},i}, t-d, a_0(Rx_{\mathbf{c},i}), a_1(Rx_{\mathbf{c},i}))$$

with $listen_{start}(s) \in \Phi\ \langle cycleOfst\rangle + (s-1)\,\Delta\ \langle slot\rangle + \Phi\ \langle slotOfst\rangle$ and $listen_{end}(s) \in listen_{start}(s) + \Delta\ \langle\Box\ rcptWin\rangle$ (see sections 3.10, 5.3 and 6.4.3).

Depending on the resulting behavior at the receivers there is a subset $D_{\mathbf{c},s} \subseteq C_t$ with

$$D_{\mathbf{c},s} = \{i \mid i \in C_t \wedge \mathrm{RES}(\mathbf{c}, i, s) = 1\}$$

A receiver $i \in C_t$ received a transmission without faults, if $i \in D_{\mathbf{c},s}$.

A transmission can lead to symmetrical fault-free or faulty behavior, or byzantine faults. Fault-free behavior is present, if $D_{\mathbf{c},s} = C_t$. If $D_{\mathbf{c},s} = \emptyset$, then the behavior is faulty, but non-byzantine. In all other cases, a faulty component causes byzantine faults.

7.2 The Impact of Single Component-Faults

In case of a single component fault in a component network $\mathrm{N} = (\mathcal{K}, \mathcal{V})$, there is at least one component $k \in \mathcal{K}$ with $\kappa(v_i, t) \leq 1$ for all $v_i \in O_k$ and $t_{start} \leq t \leq t_{stop}$. On the other side, $\phi(w_j, t) = 1$ for all $w_j \in \mathcal{I}_k$ is guaranteed, since all other components in the network behave fault-free.

In the following, the behavior of faults during a cycle is investigated. The following production from the timed grammar for TEA in chapter 5 shows the definition of a cycle.

$$cycle \rightarrow \Box\ (cycleOfst\ .\ regPart\ .\ extPart)$$

A cycle is a sequence of cycle offset, regular part and extension part. The production defines the duration of a cycle as its length, while the offset is set to [0]. This is the effect of the $\Box$-operator, so that $t_{start} = 0$ and $t_{stop} = \max \Delta\ \langle cycle\rangle$.

7.2.1 Input Driver Fault

If an input driver $k \in \mathcal{D}_r$ is the faulty component in a network, and k is connected to a controller i on channel $\mathbf{c}$, the network behavior becomes

$$\begin{aligned} \phi(Rx_{\mathbf{c},i}, t) &= \kappa(Rx_{\mathbf{c},i}, t)\, \phi(RxCh_{\mathbf{c},i}, t-d) \\ &= \kappa(Rx_{\mathbf{c},i}, t) \end{aligned}$$

at any time. All transmissions on channel $\mathbf{c}$ are affected. Therefore, the result $RES(Rx_{\mathbf{c},r}) < 1$ is possible for a receiver r at any time. This means, that r may accept or reject each message on channel $\mathbf{c}$. Byzantine faults are, therefore, possible since the result for r can be different from the result for all other receivers.

7.2.2 Gate Fault

The question, if a gate shows faulty behavior or not, depends on the intended behavior of the switch within the gate. The switch should provide the output driver access to the channel, if the intended behavior is active. This is the case during the message window, if the connected controller i on channel $\mathbf{c}$ is scheduled to access the channel, so that sender$(s, \mathbf{c}) = i$. This happens at least once in the regular part, and may be the case during the extension part.

A fault-free switch is controlled by the neighbor controller j of the sender i. To determine the time, the neighbor opens the gate, it is assumed that the neighbor starts to open the gate at time $ng_{start}(s) \in (s-1)\,\Delta\,\langle slot\rangle + \Phi\,\langle slotOfst\rangle$ and closes it at $ng_{end}(s) \in ng_{start}(s) + \Delta\,\langle\Box\,neighbor\rangle$. This means, that

$$\sigma(Pwr_{\mathbf{c},i}, t) = \begin{cases} 1 & \text{if } ng_{start}(s) \leq t \leq ng_{end}(s) \;\; \forall\ \text{sched}(s, \mathbf{c}) = i \\ 0 & \text{else} \end{cases}$$

A switch and an output driver fault are considered two different single component faults. Therefore, the two inner components of a gate are analyzed separately. In case of a switch fault, the behavior of the output driver is fault-free, so that $\phi(Pwr_{\mathbf{c},i}, t) \leq 1$ and $\phi^1(TxCh_{\mathbf{c},i}, t) = 1$.

This means, that

$$\phi^2(TxCh_{\mathbf{c},i}, t) = \sigma(Pwr_{\mathbf{c},i}, t)\,\phi(Pwr_{\mathbf{c},i}, t)$$

and

$$\begin{aligned} \phi^3(TxCh_{\mathbf{c},i}, t) &= (1 - \sigma(Pwr_{\mathbf{c},i}, t))\,(\phi(Pwr_{\mathbf{c},i}, t) + (1 - \phi(Pwr_{\mathbf{c},i}, t)) \\ &= (1 - \sigma(Pwr_{\mathbf{c},i}, t)) \end{aligned}$$

If a fault-free sender is not allowed to access the channel, so that $\sigma(Pwr_{\mathbf{c},i}, t) = 0$, then $\phi^2(TxCh_{\mathbf{c},i}, t) = 0$ and $\phi^3(TxCh_{\mathbf{c},i}) = 1$. The behavior at the output connection to the channel becomes

$$\phi(TxCh_{\mathbf{c},i}, t) = \phi^2(TxCh_{\mathbf{c},i}, t) + \phi^3(TxCh_{\mathbf{c},i}, t) = 1$$

On the other side, if the sender is allowed to send, then

$$\phi^2(TxCh_{\mathbf{c},i}, t) = \phi(Pwr_{\mathbf{c},i}, t)$$

and $\phi^3(TxCh_{\mathbf{c},i}, t) = 0$.

Therefore, a faulty switch can cause faults during the time the sender is allowed to access the channel. Since all other components behave fault-free, the faulty behavior $\phi(Pwr_{\mathbf{c},i}, t) \leq 1$ is propagated (with a channel delay) to all receivers:

$$\phi(RxCh_{\mathbf{c},r}, t) = \phi(TxCh_{\mathbf{c},i}, d_{r,i})$$

for all $r \in C_t$, so that $\phi(Rx_{\mathbf{c},r}) = \phi(RxCh_{\mathbf{c},r}, t) = \phi(TxCh_{\mathbf{c},i}, d_{r,i}) \leq 1$, because the input drivers are also fault-free.

This means, that a switch fault can cause transmission failures, if the respective sender is allowed to access the respective channel. Note, that in this model a switch fault is also promoted to the channel, if the message of a fault-free sender already passed the fault-free output driver, so that a faulty switch cannot physically affect a message. A receiver may detect a message fault, if the switch behaves faulty when the message passes the output driver. If $delay_{out}$ is the delay of the output driver, $mw_{start}(s) \in ng_{start}(s) + \Phi\,\langle msgOfst\rangle$ and $mw_{end}(s) \in mw_{start}(s) + \Delta\,\langle msg\rangle$, then a message fault occurs if

$$\prod_{t=mw_{start}(s)}^{mw_{end}(s)} \phi(TxCh_{\mathbf{c},i}, t + delay_{out}) < 1 \qquad (7.1)$$

Depending on the tolerances of the receiver, the fault can be detected.

A faulty output driver affects a message at any time, because it has similar properties as the input driver. Furthermore, it can spontaneously generate random signals. But a driver fault can only be propagated to the channel, if it is empowered by the switch, which is only the case during the message window, so that $\phi(TxCh_{\mathbf{c},i}, t + delay_{out}) = \kappa(TxCh_{\mathbf{c},i}, t + delay_{out}) + \phi((Tx_{\mathbf{c},i}, t))$ for $\sigma(Pwr_{\mathbf{c},i}, t + delay_{out}) = 1$ and a fault-free switch. In case of a single component fault the sender is also fault-free, so that

$$\prod_{t=mw_{start}(s)}^{mw_{end}(s)} \kappa(TxCh_{\mathbf{c},i}, t + delay_{out}) \leq 1 \tag{7.2}$$

The faulty behavior of the driver is propagated to the receivers, and may be detected.

7.2.3 Channel Fault

Faulty channels can corrupt a signal in a different way at different output connections. These connect the channel to the respective input driver of the receivers. Message corruption is possible at any time. This means, that every single message can become corrupted or non-corrupted for different receiving controllers. If the message is corrupted, then it might be accepted by some receivers due to measurement tolerances.

Additionally, a channel can generate random signals. It cannot generate phantom messages. If a receiver does not detect a coding error, then the decoded message content is equal to the message content sent by the sender. This includes message header and body. Otherwise a checkword error is detected.

In case of a single component fault, the network behavior is

$$\phi(RxCh_{\mathbf{c},r}, t) = \kappa(RxCh_{\mathbf{c},r}, t)$$

for all receivers $r \in C_t$ at any time, where $\mathbf{c} \in \{\mathbf{A}, \mathbf{B}\}$ is the faulty channel. Since a faulty channel can behave differently on different output connections, it can cause byzantine failures.

7.2.4 Controller Fault

A faulty controller can behave faulty on the connections to the output drivers and on the connections to the switches of the neighbors gates. The behavior can be different on different output connections. It can behave faulty at any time.

On the connection to an output driver on channel $\mathbf{c}$, the faulty controller i is able to cause faulty behavior, so that $\phi^1(TxCh_{\mathbf{c},i}, t) = \kappa(TxCh_{\mathbf{c},i}, t)\,\phi(Tx_{\mathbf{c},i}, t) = \kappa(TxCh_{\mathbf{c},i}, t)\,\kappa(Tx_{\mathbf{c},i}, t) = \kappa(Tx_{\mathbf{c},i}, t) \leq 1$. The resulting behavior at $TxCh_{\mathbf{c},i}$ is similar to the behavior of a faulty output driver.

This is also true for the behavior at connection $Guard_{\mathbf{c},i,j}$ (where j is the fault-free sender), because $\phi(Pwr_{\mathbf{c},j}, t) = \kappa(Pwr_{\mathbf{c},j}, t)\,\phi(Guard_{\mathbf{c},i,j}) \leq 1$.

Due to the architecture of a TEA node, a faulty node behaves similar to a faulty output driver of the same gate, and a faulty switch of the neighbor's gate. So, the results from section 7.2.2 apply here, also.

7.3 Controller- and Channel-Compounds

In TEA, drivers and switches share one important property: If the behavior at their output connection is faulty, then the component itself is faulty, or the component connected to the input connection is faulty, or even both. In all cases the result is similar. The behavior at the output connection becomes faulty. The reason is, that the output behavior depends only on the behavior of the input connection and the component behavior. If $k \in \mathcal{D}_g \cup \mathcal{D}_r \cup \mathcal{S}_g$ (input/output drivers and switches – see section 6.3) and $h, l \in \mathcal{K}$, the network behavior is $\phi(o_{k,l}, t) = \kappa(o_{k,l}, t)\,\phi(i_{h,k}, t)$ with $\mathcal{I}_k = \{i_{h,k}\}$ and $\mathcal{O}_k = \{o_{k,l}\}$. In case of a faulty component h at the input connection the result is $\phi(o_{k,l}, t) = \kappa(o_{k,l}, t)\,\kappa(i_{h,k}, t)$, if all other components behave fault-free. Faulty behavior at $o_{k,l}$ can now have different reasons:

- Component k behaves faulty, so that $\kappa(o_{k,l}, t) < 1 \Rightarrow \phi(o_{k,l}, t) < 1$
- Component h behaves faulty, so that $\kappa(i_{h,k}, t) < 1 \Rightarrow \phi(o_{k,l}, t) < 1$
- Both components h and k behave faulty, so that $\kappa(o_{k,l}, t) < 1 \wedge \kappa(i_{h,k}, t) < 1 \Rightarrow \phi(o_{k,l}, t) < 1$

The conclusion is, that drivers and switches do not need to be analyzed separately. These components can be put together with their adjacent component on the input connection in one fault-region. Two types of fault-regions can be identified:

1. Controller compound $Ctrl_i$ for a controller $i \in C_t$: Contains a controller and the connected output drivers and switches.
2. Channel compound $Chan_{\mathbf{c}}$ for $\mathbf{c} \in \{\mathbf{A}, \mathbf{B}\}$: Contains a channel, and all connected input drivers of all controllers.

All inputs of all drivers and switches in one compound are connected to one single component, either controller or channel. This component can behave faulty or fault-free at any of its outputs. As shown above, the output connections of the drivers and switches show faulty behavior, if this single component shows faulty behavior at the connection to the driver or switch component. If controller i in compound $Ctrl_i$ behaves faulty, on one or more of its output connections, then this is equivalent to faulty components at these connections.

Since $\kappa(Tx_{\mathbf{c},i}, t) < 1 \Rightarrow \phi(TxCh_{\mathbf{c},i}, t) < 1$ and $\kappa(Guard_{\mathbf{c},i,j}) < 1 \Rightarrow \phi(Pwr_{\mathbf{c},j}) < 1$ for channels $c \in \{\mathbf{A}, \mathbf{B}\}$ and $i, j \in C_t$, where j is the neighbor of i, it is sufficient to analyze faulty behavior of the controller i of compound $Ctrl_i$. The set of faulty output connections of a controller compound $k = Ctrl_i$ is $\mathcal{F}_{k,t} \in \mathcal{P}(O_k)$ (an element of the powerset of O_k) and, therefore $0 \leq \Upsilon_{k,t} \leq 1$.

On the other side, faulty behavior of one or more of the output drivers or switch components is a subset of the possible behavior of the controller compound with a faulty controller. If only one output connection can show faulty behavior, then $\Upsilon_{k,t} \in \left\{0, \frac{1}{4}\right\}$. These are two special cases also possible in case of a controller fault. The same is true, if more than one component shows faulty behavior. If, for example, one output driver and one switch is faulty, then $\Upsilon_{k,t} \in \left\{0, \frac{1}{4}, \frac{1}{2}\right\}$. This is also possible in case of a controller fault, where the set of values for $\Upsilon_{k,t}$ contains all possible values. This is $\Upsilon_{k,t} \in \left\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right\}$.

With these results, single faults in a controller compound can be defined in the following way: A controller compound $Ctrl_i$ behaves faulty on its outputs, if one or more components in this region behave faulty. A single fault is present in a network, if one controller region shows faulty behavior, regardless of the number of faulty components in this region. This definition is also valid in a similar way for channel compounds.

In the following, the term “single fault” refers always to a controller or channel region.

7.4 Single Controller- and Channel Compound Faults

This section summarizes the effects of a single fault. A single fault is present in the network, if at least one connection of one compound shows faulty behavior. Two different scenarios are possible. A fault present in one controller or one channel compound.

The last section showed, that a controller fault covers all other possible component faults within a controller compound, so that the results from section 7.2.4 apply also for controller compounds.

In case of a single-fault in the compound $Ctrl_i$ it is possible, that the channel is affected if controller i is the current sender and the neighbor has opened the gate, so that

$$\prod_{t=ng_{start}(s)}^{ng_{end}(s)} \phi(TxCh_{\mathbf{c},i}, t) \leq 1$$

with $\text{sched}(s, \mathbf{c}) = i$ and $\mathbf{c} \in \{\mathbf{A}, \mathbf{B}\}$. Both channels are affected in the extension part, while only one channel can be affected in the same slot during the regular part. In all other cases, the result is 1 (fault-free behavior at $TxCh_{\mathbf{c},i}$).

The second consequence of a controller compound fault is, that the neighbor can be affected, provided it is the current sender in this slot. This means, that

$$\prod_{t=mw_{start}(s)}^{mw_{end}(s)} \phi(Pwr_{\mathbf{c},j}, t) \leq 1$$

and $\text{sched}(s, \mathbf{c}) = j$ where j is the neighbor controller of i. Also, channels **A** and **B** can be affected in the same slot if the neighbor is scheduled for the extension part.

In both cases, the fault affects the connections $TxCh_{\mathbf{c},i}$ and $TxCh_{\mathbf{c},j}$ in their respective slots on the respective channels, so that the same behavior is propagated to all $RxCh_{\mathbf{c},r}$. Because TEA requires, that $a_0(Rx_{\mathbf{c},x}) = a_0(Rx_{\mathbf{c},y})$ and $a_1(Rx_{\mathbf{c},x}) = a_1(Rx_{\mathbf{c},y})$ for all $x, y \in C_t$, the result is the same for all receivers, since

$$\begin{aligned} \theta(Rx_{\mathbf{c},x}, t-d, a_0(Rx_{\mathbf{c},x}), a_1(Rx_{\mathbf{c},x})) &= \theta(RxCh_{\mathbf{c},x}, t-d, a_0(Rx_{\mathbf{c},x}), a_1(Rx_{\mathbf{c},x})) \\ &= \theta(RxCh_{\mathbf{c},y}, t-d, a_0(Rx_{\mathbf{c},y}), a_1(Rx_{\mathbf{c},y})) \\ \Rightarrow \text{RES}(\mathbf{c}, x, s) &= \text{RES}(\mathbf{c}, y, s) \end{aligned}$$

The consequence is, that a controller fault cannot cause byzantine behavior.

Channel compound faults can cause transmission faults in every slot in the cycle. The results from sections 7.2.3 and 7.2.1 apply here also. This means, that the channel compound can behave faulty at any time. Because the receivers recognize faults only in the reception window of a slot, the effect of a channel compound fault can be characterized for each slot s and receiver $r \in C_t - \{i\}$ separately in the following way:

$$\prod_{t=listen_{start}(s)}^{listen_{end}(s)} \phi(Rx_{\mathbf{c},r}) \leq 1$$

7.5 The Impact of Double-Faults

TEA allows one faulty channel compound and one faulty controller compound at the same time. A faulty channel compound $Chan_{\mathbf{c}}$ can cause transmission failures in all slots. Using the result function from section 7.1 the result for each receiver becomes

$$\text{RES}(\mathbf{c}, r, s) = \prod_{t=listen_{start}(s)}^{listen_{end}(s)} \theta(\phi(Rx_{\mathbf{c},r}, t), a_0(Rx_{\mathbf{c},r}), a_1(Rx_{\mathbf{c},r})) \in \{0, 1\}$$

in each slot $1 \leq s \leq |requestPhase| + |confirmPhase| + |extPart|$. In slot s with $\text{sched}(s, \mathbf{c}) = i$, $c \in \{\mathbf{A}, \mathbf{B}\}$ and $Ctrl_i$ is the faulty compound, then

$$\text{RES}(\mathbf{c}, r, s) = \prod_{t=listen_{start}(s)}^{listen_{end}(s)} \theta(\kappa(Rx_{\mathbf{c},r}, t)\, \kappa(RxCh_{\mathbf{c},r}, t)\, \phi(TxCh_{\mathbf{c},i}, t), a_0(Rx_{\mathbf{c},r}), a_1(Rx_{\mathbf{c},r})) \in \{0, 1\}$$

The result is the same, if $\text{sched}(s, \mathbf{c}) = j$ and j is the neighbor controller of i.

If a faulty controller or its neighbor is sending on the fault-free channel, then the results from section 7.4 apply independent from the results on the faulty channel, which means, that a symmetric and byzantine failure can occur in the same slot on different channels.

If the components show fault-free behavior, then

$$\text{RES}(\mathbf{c}, r, s) = 1$$

for all $s \in \{t \mid \text{sched}(t, \mathbf{c}) \notin \{i, j\} \wedge Chan_{\mathbf{c}} \neq \mathbf{d}\}$, and $\mathbf{d}$ is the faulty channel compound.

7.6 Faults in the Value Domain

Every message in TEA is a carrier of information. In case of a successful transfer, this information is the same as sent by the current sender in the current slot. Otherwise, the message can be identified as corrupted, because TEA uses reliable checkword mechanisms. But in case of a faulty controller there can be another kind of fault. A controller can send a message with an incorrect value. In relation to protocol-relevant information this means, that the value does not meet the requirements of the protocol.

If the transfer is successful, the receivers have to decide to accept or reject this value. Sometimes, incorrect values can be identified, if they do not match a certain expected value. This may be the identification number of the controller, a wrong serial number of a message or other information the receivers can check.

A receiver which cannot detect a wrong value, has to accept the transmission. It is possible, that some controllers can identify incorrect information, while others don't. In a situation where an information is sent twice for redundancy reasons, the first message might get lost, so that only a subset of all receivers get the information and can compare it to the same information in the second message sent afterwards. These controllers are able to detect contradicting values, while controllers who received only one message cannot. In any case, the value sent by the faulty sender is incorrect.

Assume that $\text{correct}(mess_{i,k})$ is the correctness of the k^{th} bit in the received message $mess_i$ sent by controller i. The bit $mess_{i,k}$ is correct, if $\text{correct}(mess_{i,k}) = 1$, otherwise $\text{correct}(mess_{i,k}) = -1$. Incorrect bits are only possible, if the sender i is faulty. In fact, incorrect information is only possible, if there is a faulty controller component within a compound.

The validity of a message depends on the correctness and on the availability of an information. An information is available, if there were no transmission faults, so that

$$\text{valid}(mess_{i,k}, r, s, \mathbf{c}) = \text{RES}(\mathbf{c}, r, s)\,\text{correct}(mess_{i,k})$$

Provided that $i, j \in C_t$ are controllers, j is the neighbor of i and $\text{sched}(s, \text{c}) = i$, then the following is true for TEA:

- If $Chan_{\mathbf{c}}$ is not faulty (which means it contains no faulty components), and $Ctrl_i$ and $Ctrl_j$ are fault-free, then $\text{valid}(mess_{i,k}, r, s, \mathbf{c}) = 1$.
- If $Ctrl_i$ is faulty, then $\text{RES}(\mathbf{c}, r, s) \in \{0, 1\}$ and $\text{correct}(mess_{i,k}) \in \{1, -1\}$, so that $\text{valid}(mess_{i,k}, r, s, \mathbf{c}) \in \{1, 0, -1\}$.
- Otherwise, either $Ctrl_j$ or $Chan_{\mathbf{c}}$ is faulty or both, then $\text{RES}(\mathbf{c}, r, s) \in \{0, 1\}$ and $\text{correct}(mess_{i,k}) = 1$ (sender i is fault-free), so that $\text{valid}(mess_{i,k}, r, s, \mathbf{c}) \in \{1, 0\}$.

7.7 Summary

The purpose of this chapter was to show, how the protocol operation is affected in the case of single- and double-faults. It could be shown, that transmissions might be affected by component faults. A formal definition of fault-free and faulty transmission was given.

One result was, that some combinations of multiple component faults could be regarded as single-compound fault. Each compound has one central component, either a controller or a channel. A faulty controller or channel can cause every possible faulty behavior in their respective compound.

Since faulty components can behave also fault-free, it is only necessary to analyze double-compound faults because they cover every possible behavior. In fact, TEA allows $|C_t| + 6$ component faults at the same time. These are the input drivers in one channel compound ($|C_t|$ is the number of controllers, and also the number of all input drivers on one channel), then the channel itself. Additionally the two input drivers and switches in the controller compound and the controller must be taken into account.

In case of double-faults, all transmissions on one channel region can be become affected. Additionally, there may be faulty transmissions on the fault-free channel, either in the slot of the controller belonging to the faulty controller region, or the neighbor of that controller, depending on the schedule. Failures on the fault-free channel are always symmetric, while a faulty channel can show byzantine behavior. The TEA protocol must assure that dynamic arbitration is possible under this most pessimistic condition. This is assured by the agreement protocol described in the next chapter.

8 Dynamic Allocation of Slots in TEA

The TEA protocol provides dynamic allocation of slots in the extension part of a communication cycle. The communication controllers can dynamically request sending rights in the regular part. In contrast to the extension part, the access rights in the regular part are pre-configured and static during runtime. To determine a schedule for the extension part, all controllers must agree on a set of senders in the regular part of the cycle. Additionally, the agreement algorithm must come to correct results even in the case of faults.

The following section presents the agreement algorithm which is used to determine a schedule for the dynamic extension part in the same cycle than the preceding regular part. A proof of correctness is given is section 8.2. The algorithm and the proof have been originally published in [Lis04].

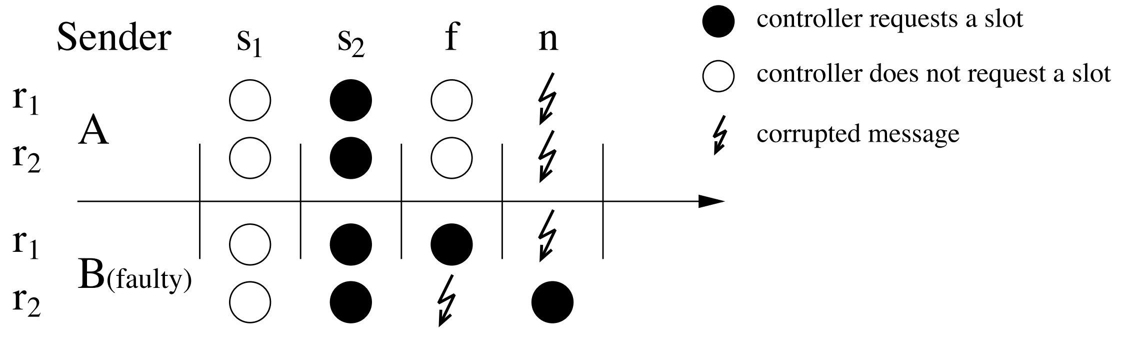
Since the selection of senders is the first step in building a schedule for the extension part, this method was also named *agreement based scheduling* in [Lis05a].

8.1 Determining a Set of Senders for the Extension Part

The goal of the agreement algorithm is to gain a unique set of senders for the dynamic part for all controllers in the network. This should be accomplished after the preceding regular part in the same cycle finished. In order to tolerate double-faults, it is not sufficient for each controller to send a request on both channels as shown in figure 8.1. This naive implementation tolerates single-faults, but may fail in case of double-faults. In the following, the terms “channel fault” and “controller fault” refer to the respective compounds, with the exception of value failures which can only be caused by faulty controller components.

In case of a single-fault with a faulty channel compound, each controller receives the same message on the fault-free channel. Since the controller sends on both channels the same request bit, all controllers come to the same conclusion, regardless of they receive the message on the faulty channel or not. The channel cannot alter the request bit without making the message invalid, because the message would not pass the checkword test.

In case of a faulty controller compound and two fault-free channels, the messages from the faulty controller and its neighbor can become corrupted, too. If the messages on both channels are affected, then a default value for the



Sender
s1
s2
f
n
r1
r2
A
r1
r2
B(faulty)
controller requests a slot
controller does not request a slot
corrupted message

Figure 8.1: In contrast to receiver r_2, receiver r_1 sees a contradicting requests of controller f. On the other side, receiver r_1 does not receive any request of controller n at all.

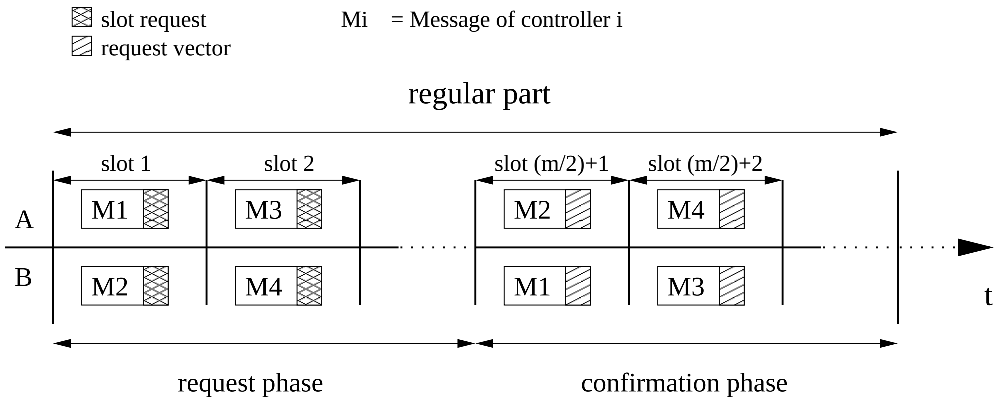


Figure 8.2: The regular part of a TEA cycle is split into request and confirmation phase. Both phases have equal length. In each slot two differenct controllers are sending on the different channels.

request must be assumed. Otherwise, one message could be received and the request value is taken from it. The faulty controller is the only one who can cause message faults. This means, that it sends different values for the request bit on both channels. This is non-critical in a single-fault situation, because this is seen by any controller, and a default value can be selected.

On the other side, the naive algorithm can fail in case in the presence of double-faults. If, for example, a faulty controller sends a corrupted message on the fault-free, but a non-corrupted on the faulty channel (see figure 8.1). The channel can behave byzantine, so some controllers can get the request bit from the message received on the faulty channel, while others cannot. It is possible, that the controllers do not agree on the same value, the value of the request bit might not match the default value, which must by chosen by the controllers which didn't receive a (non-corrupted) message on both channels.

To solve these problems, the protocol requires all controllers to confirm all requests which could be received. Therefore, the regular part is split into two phases, the request and the confirmation phase. To give all controllers the opportunity to send in both phases, without the need to double the length of the regular part, each slot is shared by two controllers sending on different channels. Since all controllers must be able to send at least once on two channels, the controllers should send on a different channel in the confirmation phase than in the request phase. Both phases have equal length. If m is the number of controllers, then $\frac{m}{2}$ is the minimum number of slots in the request and confirmation phase. This arrangement is shown in figure 8.2. The number of controllers in a TEA network is always even as mentioned in section 6.2, because every controller must be guarded by a neighbor controller on the same node. Although additional slots are possible, a regular part with only m slots should be assumed.

In both phases the messages contain different protocol relevant informations. In the request phase the controller sends one request bit. This has either the value `request` or `non_request`. The messages in the confirmation phase contain a vector $confirm\_vector$ with one entry per controller, where the entry holds either the value of the request bit received or the value `unknown_request`. The last value is sent in the case of message corruption when the controller could not determine the value of the request bit.

Each controller collects all confirmation vectors in the request matrix *request_matrix*. This is a $m \times m$ matrix where each column is filled with one confirmation vector. If a message of one controller could not be received in the confirmation phase, then the respective row is filled with the value `unavailable`. Otherwise, the entry $request\_matrix(i, j)$ contains the request of node j received in the request phase by node i.

The algorithm uses the following global declarations:

```
m constant Integer := 8; — number of controllers (8 is minimum in TEA)
id constant Controller := 1; — the controller performing the algorithm
                             — (number 1 in this case)
type Channel is (A, B);
type Matrix_Value is (request, non_request, unknown_request, unavailable);
subtype Confirm_Value is Matrix_Value range request .. unknown_request;
subtype Request_Value is Confirm_Value range request .. non_request;
type Confirm_Vector is array (1 .. m) of Confirm_Value;
type Request_Matrix is array (1 .. m, 1 .. m) of Matrix_Value;
req_val : Request_Value;
confirm_vector : Confirm_Vector;
request_matrix : Request_Matrix;
result_vector : array (1 .. m) of Request_Value;
```

The values for the constants *m* and *id* are example values. *m* is a network global parameter, while *id* is different for each controller. It's value lies in the range from 1 to *m*. The array *result_vector* contains the result of the agreement algorithm (`request` or `non_request`). The array is used in chapter 10 for building a schedule for the extension part. The variable of type Controller represents a single controller. This can be a synonym for the Integer type, for example, which is assumed here.

8.1.1 The Request Phase

The processing of the regular part is shown in the following procedure below. The main function of the request part is to build the confirmation vector *confirm_vector*. The value of the variable *slot* is the current slot, ranging from 1 to $\frac{m}{2}$. The function `sender` has been introduced in section 5.4. It returns the controller which is scheduled for the current slot on the given channel. Since TEA requires, that each controller sends once in the request phase either on channel **A** or **B**, there is always one $slot \in \{1, \ldots, \frac{m}{2}\}$ and $\mathbf{ch} \in \{\mathbf{A}, \mathbf{B}\}$ with $\texttt{sender}(slot, \mathbf{ch}) = i$ for all $i \in C_t$.

```
procedure request_phase is
  current_sender : Controller;
begin
  — start the request phase
  for slot in 1 .. $\frac{m}{2}$ loop
    synch_to_start_of_slot;
    — Channels A and B are processed in parallel
    for channel in A .. B loop
      current_sender := sender(slot, channel);
      if current_sender = id then
        — this controller is sender in the current slot on this channel
        — set req_val depending on the needs of the controller
        — prepare message with payload and header
        prepare_message(msg);
        if req_val = request then
          set_request_bit(msg);
          confirm_vector(id) := request;
        else
          clear_request_bit(msg);
          confirm_vector(id) := non_request;
        end if;
```

```
121           encode_message(msg);
122           send_to_channel(msg, channel);
123         else
124           msg := receive_from_channel(channel);
125           verify_checkword(msg);
126           if corrupted(msg) then
127             confirm_vector(current_sender) := unknown_request;
128           elsif request_bit_set_in(msg) then
129             confirm_vector(current_sender) := request;
130           else
131             confirm_vector(current_sender) := non_request;
132           end if;
133         end if;
134       end loop; -- continue with next channel
135     end loop; -- continue with next slot
136   end request_phase;
```

For each slot, the current sender is determined for both channels. Channels **A** and **B** must be processed at the same time. In line 109 it is checked, if the local controller is the current sender. If so, then the message must be prepared. Therefore, the request bit is encoded within the message and the message is send to the current channel (lines 110 to 122). In lines 116 and 119, the entry in the confirmation vector is set depending on the request bit of the current controller (the current sender).

If the controller is a receiver on the current channel, then the received message must be decoded and the request bit must be set in the confirmation vector depending on the result (lines 124 to 132). If the received message is corrupted, which means that it could not be decoded, then the entry of the sender in the confirmation vector is set to `unknown_request`.

At the end of the request phase, each entry in the confirmation vector is set, because every controller sends once in the request phase. For a fault-free controller sending on the fault-free channel, the entry is set to either `request` or `non_request`. The value `unknown_request` is only possible, if the sender is faulty, the neighbor of the faulty controller or is sending on the faulty channel. Every fault-free controller (including the neighbor of the faulty controller) sets its own value correctly.

8.1.2 The Confirmation Phase

The confirmation phase begins with the exchange of the vectors *confirm_vector* and building of the matrix *request_matrix*. This phase starts in slot $\frac{m}{2}+1$ and stops after slot m. The requirements for the function `sender` is valid for the confirmation phase, too, except that *slot* is now a value between $\frac{m}{2}+1$ and m. Additionally, every controller sends on the other channel than in the request phase.

Figure 8.3 shows an example matrix. Each column contains a confirmance vector received on the respective channel from the respective sender. Each row contains the original request as received in the request phase by the respective sender, or an `unknown_request` value. Other values are only possible, if the receiver failed to receive a message or the value has been sent by the faulty controller. In figure 8.3 the controller could not receive the confirmance vectors from senders s_4, s_5 and s_7. The messages from s_5 and s_6 became corrupted by a channel fault on channel **B**. The cause for the corruption of the message from s_4 on a fault-free channel is a controller fault. The faulty controller s_3 sent arbitrary values, for example a `request` value instead of the correct `non_request` value for controller s_2.

The following algorithm shows the processing of the confirmation phase.

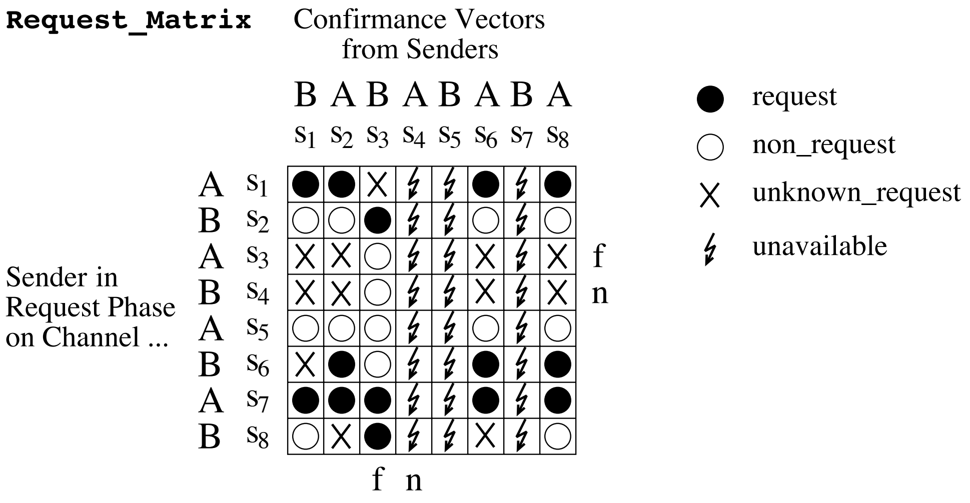


Figure 8.3: A possible request matrix for 8 controllers. The columns are the confirmance vectors received in the confirmation phase. The controllers s_3 and s_4 are the faulty controller f and its neighbor n. Channel **B** is the faulty channel.

```
procedure confirmation_phase is
  current_sender : Controller;
begin
  — start the confirmation phase
  for slot in $\frac{m}{2}+1$ .. m loop
    synch_to_start_of_slot;
    — channels A and B are processed in parallel
    for channel in A .. B loop
      current_sender := sender(slot, channel);
      if current_sender = id then
        — this controller is sender in the current slot on this channel
        — copy confirm_vector to request matrix
        — for usage after the confirmation phase
        request_matrix(id) := confirm_vector;
        — generate header and content
        prepare_message(msg);
        pack_confirm_vector_into_message(msg);
        encode_message(msg);
        send_to_channel(msg, channel);
      else
        msg := receive_from_channel(channel);
        verify_checkword(msg);
        if corrupted(msg) then
          for j in 1 .. m loop
            request_matrix(current_sender, j) := unavailable;
          end loop;
        else
          request_matrix(current_sender) := get_confirm_vector_from_message(msg);
        end if;
      end if;
```

```
230       end loop; — continue with next slot
231     end loop; — continue with next channel
232   end confirmation_phase;
```

The procedure confirmation_phase works similar to the request_phase procedure. If the current controller is the sender in the current slot on the current channel, then the confirmation vector *confirm_vector* is sent to the channel (line 210 to 218). The controller copies its own confirmation vector in the row of its request matrix which belongs to itself (line 213). Otherwise, the message is received from the channel and verified. If the message is non-corrupted, and the confirmation vector in the message could be decoded, then it is copied in to the row of the sender in the request matrix. Otherwise, this row is filled with the value `unavailable` (lines 220 to 228).

Line 216 shows that the confirmation vector is *packed* into the message. Each entry in the confirmation vector can have one of three values (`request`, `non_request` and `unknown_request`). The consequence is, that at least two bits are needed to encode one entry. With a minimum number of eight controllers, 16 bits have to be reserved in the message ($2m$ bits in general). But the message length is limited, so that the bits are, on the other side, a valuable resource. Therefore, the entries for each two controllers are grouped into a message into three bits. In general, if there are m controllers, then the confirmation vector can be encoded in $3\,\frac{m}{2}$ bits, decreasing the overhead significantly. There is no additional space needed for the one bit value in the request phase, because one bit reserved for the confirmation vector can be used.

8.1.3 An Algorithm for Agreement on the Schedule in the Extension Part

After the confirmation phase, the regular part ends with determining the senders for the extension part. In the fault-free case, the matrix of each controller contains one vector per controller with the values `request` and `non_request`. In case of a channel fault, the vectors, from the controllers sending on the faulty channel in the confirmation phase, may be set to `unavailable`. The number of non-available vectors may differ for each controller, since a channel can behave byzantine. The vectors available can contain the value `unknown_request`, if the original sender in the request phase sent on the faulty channel region. Additionally, if one controller region is faulty, then there may be one or two symmetric faults, which means that the entire vectors coming from the faulty controller or its neighbor in the matrix may not be available at the receivers, even if it sent on the fault-free channel. If the vector from the faulty controller is available, then the values may be wrong.

Therefore, a faulty controller and its neighbor sending on the fault-free channel in the confirmation phase is the worst case, since it can behave faulty and there can be wrong values.

Each controller has to perform an agreement algorithm, using the information gathered in the regular part. The result must be the same for all fault-free controllers. The function majority(c, *channel*) returns the majority of requests in the column for controller c of the matrix. It tolerates one misleading request from a faulty-controller. This is the reason why the function can return a valid request only, if at least two controllers vote for one particular value, either `request` or `non_request` (see line 320 and lemma 3 in section 8.2). While counting the requests in the respective column for controller c in the matrix, the values `unavailable` and `unknown_request` are ignored. The function returns `unknown_request`, if none of the counters for the two values `request` and `non_request` reaches at least two. Each channel is handled separately, because the results of the faulty and fault-free channel must be handled differently. For details see section 8.2.

```
300   — Helper function: scheduled_for_channel returns true, if a controller
301   — (first argument) is scheduled for a specific channel (second argument)
302   — in the confirmation phase.
303   function scheduled_for_channel(c : Controller; channel : Channel)
304       returns Boolean;
305
306   function majority(c : Controller; channel : Channel) returns Conform_Value is
307     request_counter       : Integer := 0;
```

```
308     non_request_counter    : Integer := 0;
309   begin
310     — for each value received from remote controllers for c
311     for i in 0 .. m loop
312       if scheduled_for_channel(i, channel) then
313         if request_matrix(i,c) = request then
314           request_counter := request_counter + 1;
315         elsif request_matrix(i,c) = non_request then
316           non_request_counter := non_request_counter + 1;
317         end if;
318       end if;
319     end loop;
320     if max(request_counter, non_request_counter) < 2 then
321       return unknown_request;
322     elsif request_counter > non_request_counter then
323       return request;
324     else
325       return non_request
326     end if;
327   end majority;
```

The following code builds the vector for the schedule in the extension. A controller gains the permission to send in the extension phase, if the value in the vector for this controller is `request`. The default value for a controller, when the scheduling request could not be determined, is `non_request`. This is only an example setting. The other option is to allow each controller to send in the case of doubt, that is if the original request is not available. The code will be analyzed in section 8.2. The function count_unknown (see lines 406 to 410) returns the number of entries set to `unknown_request` for a controller on a given channel. If this value is greater than one (line 412) for the channel a controller is scheduled in the confirmation phase, the result is not trustworthy, and the default value must be applied. The reason is, that, if the number of `unknown_request` entries is too high, the controller has sent on the faulty channel in the request phase. A scenario is possible where two receivers get different majorities on the faulty channel, while the majority on the fault-free channel is `unknown_request`. On the other side, it is clear in such a situation, that the controller is sending on the fault-free channel in the confirmation phase, and slot reservation must be rejected if there are too little information available to guarantee agreement.

```
400   for i in 0 .. m loop
401     declare
402       maj_on_A : Confirm_Value := majority(i, A);
403       maj_on_B : Confirm_Value := majority(i, B);
404       unknown_on_channel : Integer := 0;
405     begin
406       if scheduled_for_channel(i, A) then
407         unknown_on_channel := count_unknown (i, A);
408       else
409         unknown_on_channel := count_unknown (i, B);
410       end if;
411       if (maj_on_A /= unknown_request or maj_on_B /= unknown_request then
412         if unknown_on_channel > 1 then
413           — set result for controller i in the result array o
414           result_vector(i) := non_request;
415         elsif maj_on_A = request and maj_on_B = request or
416               maj_on_A = request and maj_on_B = unknown_request or
417               maj_on_A = unknown_request and maj_on_B = request then
418           result_vector(i) := request;
```

```
        else
          result_vector(i) := non_request;
        end if;
      end if;
    end;
  end loop;
```

After the set of senders is known, a schedule is determined in the next step. This is the topic of chapter 10.

8.2 Proof of Correctness

The result of the algorithm depends on the entries in the *request_matrix*. A value can be one of `request`, `non_request`, `unknown_request` or `unavailable`. While the value `unavailable` is the result of a corrupted message (line 224 in the procedure confirmation_phase), the others origin from a sender, which is echoing what it received in the request phase. If the sender sends something else than it received, the value is regarded as incorrect as defined in section 7.6. It is reasonable to assume, that all values sent in the request phase are correct, since at that time all values are possible. Although it is possible, that a faulty sender sends no value at all (for example, in case of a random signal or fail-silent behavior), it should be assumed, that a correct value is corrupted during the process of sending on the output connections. The result remains the same: In case, for example, of a faulty output driver in the respective controller compound, a correct value has been corrupted by the driver. In case of a faulty controller in the same compound, an already corrupted message is sent. The correctness of the content of the message is is irrelevant in both cases.

A slot s belongs to the request phase, if $s \in S = \{1, \dots, \frac{m}{2}\}$, otherwise the slot is part of the confirmation phase and $s \in \overline{S} = \{\frac{m}{2} + 1, \dots, m\}$. The set of controller can be separated into the controllers scheduled for sending in the request phase on the fault-free channel and the set of controllers sending on the faulty channel in the same phase. If $\mathbf{c}$ is the fault-free channel, then $R = \{i \mid i = \mathtt{sender}(s, \mathbf{c}) \ \wedge \ s \in S\}$ and $\overline{R} = C_t - R$.

In the remainder of this section it will be necessary to compare different entries in the confirmation vectors and request matrices of different controllers. It is shown that different receivers agree on the result for one sender in the request phase under different circumstances. This could mean, that it sends on the fault-free or faulty channel or is the faulty controller or its neighbor. As stated out above, a value sent in the request phase is never incorrect. This means, that if rq^c is the request bit sent by controller c (*req_val* in the controller with $id = c$ – see line 19), then $cv_c^r = \text{valid}(rq^c, r, s, \mathbf{c}) \in \{0, 1\}$ for $c = \mathtt{sender}(s, \mathbf{c})$, $s \in S$ and r is a receiver (see section 7.6). The variable cv_c^r is the validity of the received bit in the confirmation vector of receiver r. In the algorithm, this bit is stored in $confirmation\_vector(c)$ in the controller with $id = r$ in lines 126 to 132. If $cv_c^r = 0$, then $confirmation\_vector(c) = \text{unknown}$, otherwise $confirmation\_vector(c)$ is the value of *req_val* of controller c.

Each entry $rm_{r,c}^t \in \{1, 0, -1, \times\}$ reflects the entry *request_matrix*(r, c) of the controller with $id = t$ and is defined in the following way:

$$rm_{r,c}^t = \begin{cases} 1 & \text{if valid}(cv_c^r, t, s_r, \mathbf{c}_r) = 1 \\ 0 & \text{if valid}(cv_c^r, t, s_r, \mathbf{c}_r) = 1 \wedge \text{valid}(rq_r^c, r, s_c, \mathbf{c}_c) = 0 \\ \times & \text{if valid}(cv_c^r, t, s_r, \mathbf{c}_r) = 0 \\ -1 & \text{if valid}(cv_c^r, t, s_r, \mathbf{c}_r) = -1 \end{cases}$$

Channel $\mathbf{c}_c$ and slot s_c are the channel and slot the controller c sent in the request phase. Similarly, the slot and channel used by the sender r in the confirmation phase are s_r and $\mathbf{c}_r$. The value $rm_{r,c}^t = \times$ shows the case, that the controller t received a corrupted message in the confirmation phase. Then, the respective entry in the *request_matrix* is set to `unavailable` in line 224. In the cases $rm_{r,c}^t = 1$ and $rm_{r,c}^t = 0$, the controller r sent the value it received in the request phase correctly. The difference between both cases, is what the sender r received (and therefore sends). If $rm_{r,c}^t = 1$, then r received the correct value from c and sent it correctly, which means that *request_matrix*(r, c) = *req_val*. Otherwise, the entry is set to unknown. In the last case, the entry in the matrix can be any value, but `unavailable` or the correct one.

In the following f is the faulty controller, and n is the neighbor of f on the same node.

Lemma 1 *For any receiver $r \in C_t$, $cv^r_c = 1$, if $c \in R$ and c is neither n nor f. The number of correctly received request is at least $\frac{m}{2} - 2$.*

If $c \in R$ and c is fault-free, then $cv^r_c = \text{valid}(rq^c_r, r, s, \mathbf{c}) = \text{correct}(rq^c) RES(\mathbf{c}, r, s) = 1$, because $\text{correct}(rq^c) = 1$ is always true in the request phase, and $RES(\mathbf{c}, r, s) = 1$, if the channel compound is fault-free and neither n nor f have access to the channel (see chapter 7). In the worst case, f and n are in R. In this case, the above is true for at least $|R - \{n, f\}| = \frac{m}{2} - 2$ controllers.

After the confirmation phase passed the row rm^t_r is set to $\times$, if faults led to message corruption. As a consequence, $\frac{m}{2} - 2$ rows in the *request_matrix* are not set to `unavailable` for the same reason as in lemma 1.

Lemma 2 *For any $t \in C_t$ the entry $rm^t_{r,c} \in \{1, \times\}$, if $c \in R - \{n, f\}$ and $r \neq f$.*

As shown in lemma 1, $cv^r_c = 1$. If $r \in R - \{f\}$, then $rm^t_{r,c} \in \{\times, cv^r_c\}$ because r sent on the faulty channel. If $r \notin R$, then $rm^t_{r,c} = cv^r_c$.

Lemma 3 *The function* majority$(c, channel)$ *returns req_val for the fault-free channel channel, and req_val or* unknown *for the faulty channel, for all controllers $c \in C_t$. In the case of the fault-free channel the result is the same for all receivers t.*

For $r \in \overline{R} - \{n, f\}$ (r is scheduled on the fault-free channel in the confirmation phase, which means that *channel* is set to the fault-free channel), the value $rm^t_{r,c} = 1$, if $c \in R - \{n, f\}$. This conclusion can be derived from lemma 1 and 2. Since every receiver sees this value at least $\frac{m}{2} - 2$ times, all receivers agree on the correct value *req_val*. Lemma 2 shows also, that $rm^t_{r,c} \in \{1, \times\}$, if $r \in R - \{f\}$. Since the value $\times$ is not counted in majority$(c, channel)$, the result is either *req_val*, if the value 1 appears more than once in rm^t_r, c, for all $r \in \overline{R} - \{n, f\}$ or `unknown_request`, otherwise. In case of $c \in \overline{R}$ the value in the matrix is $rm^t_{r,c} \in \{0, 1, \times\}$, or $rm^t_{r,c} \in \{0, 1, -1, \times\}$ for $r = f$. For $r \in \overline{R}$ (r is sending on the fault-free channel in the confirmation phase), the result is the same for all receivers, if r is fault-free, and for n and f, because both behave symmetrical by definition. For that reason, all controllers agree on the same value on the fault-free channel. The last case where $c \in \overline{R}$ and $r \in R$, the situation is similar to the case $c \in R$. Since the channel behaves byzantine, the result can be different for different controllers, but not the incorrect one. This is also true, in case of n and f, because an incorrect value appears not more than once (can only be sent by f), and is, therefore, outvoted in line 320.

Theorem 1 (Agreement) *The result schedule is the same for every receiver $t \in C_t$. In addition, it can be guaranteed for all $c \in R - \{f, n\}$ that the slot is reserved if it was requested, and not reserved if not requested.*

According to lemma 3, the function *majority*$(c, channel)$ returns the same value for all t if *channel* is fault-free. The value $rm^t_{r,c} = 0$ can only appear once (for $r = f$), for $c \in R$ in (lemma 1). As a result, if *majority*$(c, channel)$ returns unknown for a $c \in R$, c passes the test in line 412. Since there is always a return value other than unknown on the fault-free channel (lemma 3), *result_schedule*(c) is set to the correct value in lines 417 to 421. If $c \in \overline{R} \cup \{f, n\}$, than it is possible, that *majority*$(c, channel)$ returns unknown, because at least two entries in the matrix are set to unknown on the fault-free channel. This is always the channel c is sending on. Such a situation can become critical. In this case, the test in line 412 leads to the default value for any t. Since this happens on the fault-free channel, this situation is visible to any t. In the other cases, there can be a majority on the fault-free channel or the result for both channels is unknown. For the first case the test in line 412 passes, because this is not the channel, c is sending on. In either case c is handled the same way as if it would be from R.

Controller n is handled as a special case of $c \in \overline{R}$, because $cv_n^r = 0$ is possible for any r, if it is blocked by f. Otherwise it behaves like any other fault-free controller. Controller f behaves either like a normal controller, like n or it sends an incorrect value in the confirmation phase, in which case f is outvoted in majority(c, *channel*), or the function returns unknown.

It is up to controller n to prevent f from sending if necessary, so that the faulty controller cannot corrupt other messages than from n. This was the result of chapter 7.

9 Timing of a TEA-Network

Since TEA is a real-time protocol, the time that every action takes must be predictable. The timed grammar developed in chapter 5 can be used to determine the timing of the TEA protocol. Section 9.1 lists the preconditions that must be met before the grammar can be evaluated. The following section derives the dependencies between the parameters in the grammar.

The chapter concludes with example configurations and a case study.

9.1 Preconditions

In order to evaluate the timed grammar developed in section 5.3, the desired number, of clock units per message and the number of slots in the regular part and the extension part must be known. Therefore, the grammar must be parameterized. Some parameters are constrained, while others depend on the user-defined configuration. The number of slots, for example, is application-specific. Table 9.2 lists the parameters and their constraints. The table also shows, that most of the parameters depend on the application. In case of $|slotOfst|$ the value can be derived from the grammar, if all other parameters are given. This value is an important criterion for measuring efficiency. This topic will be discussed in section 9.3.

9.2 Evaluation of the Timed Grammar for the TEA Protocol

In this section, the grammar will be evaluated. The goal is to derive a formula for each entity of each action depending on the parameters described in table 9.2. Since each partition contains only one production, and the cardinality of the entities of the atomic actions is one, each entity for each action will contain not more than one view. Therefore, the following formulas are given for one view per action, only. This view is the only element in the resulting entity for that action. In the following, this is not written explicitly. The evaluation starts bottom up with the atomic actions. The full grammar can be found in appendix A. The evaluation is done as described in 4. An explanation for the grammar can be found in 5.

Productions 5.8, 5.11 and 5.6 There are three actions depending only on clock offsets. These are *slotOfst*, *msgOfst* and *cycleOfst*.

$$\begin{aligned}
\Phi\,\langle slotOfst\rangle &= |slotOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle slotOfst\rangle &= 0\\
\Phi\,\langle msgOfst\rangle &= |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle msgOfst\rangle &= 0\\
\Phi\,\langle cycleOfst\rangle &= |cycleOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle cycleOfst\rangle &= 0
\end{aligned}$$

with the constraint $|slotOfst| = |msgOfst|$, so that

$$\Phi\,\langle msgOfst\rangle = \Phi\,\langle slotOfst\rangle$$

Parameter	Meaning	Constraint	Comment
$\lvert extPart \rvert$	number of slots in the extension part	—	application-dependent
$\lvert reqPhase \rvert$	number of slots in the request phase	$\geq \frac{\lvert C_t \rvert}{2}$	application-dependent
$\lvert confirmPhase \rvert$	number of slots in the confirmation phase	$\lvert confirmPhase \rvert = \lvert reqPhase \rvert$	application-dependent
$\lvert slotOfst \rvert$	maximum clock units needed to overcome maximum differences between clocks at the beginning of a slot	—	
$\lvert rcptBuffer \rvert$	duration of reception buffer in clock units	$\min \Delta \langle rcptBuffer \rangle \geq \max \Phi \langle \texttt{delay} \rangle$	application-dependent
$\lvert msgOfst \rvert$	maximum clock units needed to overcome maximum differences between clocks before sending a message	$\lvert msgOfst \rvert = \lvert slotOfst \rvert$	
$\lvert msgHdr \rvert$	length of message header	—	application-dependent
$\lvert msgBdy \rvert$	length of message body	—	application-dependent
$\Phi \langle \texttt{delay} \rangle$	physical channel delay	—	physical property of the channel
$\Delta \langle \texttt{clkUnit} \rangle$	clock difference	—	depends on maximum clock skew r
$\Phi \langle \texttt{clkOfst} \rangle$	clock difference	$\Phi \langle \texttt{clkOfst} \rangle = \Delta \langle \texttt{clkUnit} \rangle$	depends on maximum clock skew r
$\gamma_f, \gamma_s, \hat{t}$	minimum ($\gamma_f \hat{t}$) and maximum ($\gamma_s \hat{t}$) length of clock unit. The nominal length of a clock tick is $\hat{t}$ (in ns, for example)	$\gamma_f = 1 - r, \gamma_s = 1 + r$	hardware-dependent

Table 9.2: The parameters to be set for the TEA protocol and their constraints.

Productions 5.13 and 5.14 The following actions are only depending on clock units.

$$\begin{aligned}
\Phi\ \langle msgHdr\rangle &= 0\\
\Delta\ \langle msgHdr\rangle &= |msgHdr|\,\Delta\ \langle \texttt{clkUnit}\rangle\\
\Phi\ \langle msgBdy\rangle &= 0\\
\Delta\ \langle msgBdy\rangle &= |msgBdy|\,\Delta\ \langle \texttt{clkUnit}\rangle\\
\Phi\ \langle rcptBuffer\rangle &= 0\\
\Delta\ \langle rcptBuffer\rangle &= |rcptBuffer|\,\Delta\ \langle \texttt{clkUnit}\rangle
\end{aligned}$$

Productions 5.12, 5.16, 5.10 and 5.15 The next goal is to determine the ⟨*sender*⟩ and ⟨*neighbor*⟩ views.

$$\begin{aligned}
\Phi\ \langle msg\rangle &= \Phi\ \langle msgHdr\rangle = 0\\
\Delta\ \langle msg\rangle &= \Delta\ \langle msgHdr\rangle + \Phi\ \langle msgBdy\rangle + \Delta\ \langle msgBdy\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\ \langle \texttt{clkUnit}\rangle\\
\Phi\ \langle msgWindow\rangle &= \Phi\ \langle msg\rangle = 0\\
\Delta\ \langle msgWindow\rangle &= \Delta\ \langle msg\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\ \langle \texttt{clkUnit}\rangle
\end{aligned}$$

The views for the *sender* and *neighbor* actions can be given, now.

$$\begin{aligned}
\Phi\ \langle sender\rangle &= \Phi\ \langle msgOfst\rangle + \Phi\ \langle msg\rangle\\
&= |msgOfst|\,\Phi\ \langle \texttt{clkOfst}\rangle\\
\Delta\ \langle sender\rangle &= \Delta\ \langle msg\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\ \langle \texttt{clkUnit}\rangle
\end{aligned}$$

Below, $\Phi\ \langle neighbor\rangle$ becomes 0 and the leading offset becomes part of the duration of *neighbor*. This is the consequence of the $\Box$–operator.

$$\begin{aligned}
\Phi\ \langle neighbor\rangle &= 0\\
\Delta\ \langle neighbor\rangle &= \Phi\ \langle msgOfst\rangle + \Phi\ \langle msgWindow\rangle + \Delta\ \langle msgWindow\rangle + \Phi\ \langle msgOfst\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\ \langle \texttt{clkUnit}\rangle + 2\,(|msgOfst|\,\Phi\ \langle \texttt{clkOfst}\rangle)
\end{aligned}$$

$\Delta\ \langle neighbor\rangle$ can be simplified, because $\Phi\ \langle \texttt{clkOfst}\rangle = \Delta\ \langle \texttt{clkUnit}\rangle$:

$$\Delta\ \langle neighbor\rangle = (|msgHdr| + |msgBdy| + 2\,|msgOfst|)\,\Delta\ \langle \texttt{clkUnit}\rangle$$

Productions 5.17 and 5.18 The physical transmission over the channel to the receiver is represented by the actions *gate* and *ch*. It must be assured that the message passes the gate, completely.

$$\begin{aligned}
\Phi\,\langle gate\rangle &= \max\{\Phi\,\langle sender\rangle\,, \Phi\,\langle neighbor\rangle\}\\
&= \max\{|msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\,, 0\}\\
&= |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle gate\rangle &= \max\{0, \min\{\Phi\,\langle sender\rangle + \Delta\,\langle sender\rangle\,,\\
&\quad \Phi\,\langle neighbor\rangle + \Delta\,\langle neighbor\rangle\} - \Phi\,\langle gate\rangle\}\\
&= \max\{0, |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
&\quad + (|msgHdr| + |msgBdy|)\,\Delta\,\langle \mathtt{clkUnit}\rangle - \Phi\,\langle gate\rangle\}\\
&= (|msgHdr| + |msgBdy|)\,\Delta\,\langle \mathtt{clkUnit}\rangle\\
\Phi\,\langle ch\rangle &= \Phi\,\langle \mathtt{delay}\rangle + \Phi\,\langle gate\rangle\\
&= \Phi\,\langle \mathtt{delay}\rangle + |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle ch\rangle &= \Delta\,\langle gate\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\,\langle \mathtt{clkUnit}\rangle
\end{aligned}$$

Productions 5.20, 5.19 and 5.22 To determine the view of the *receiver* action the reception windows' view is needed.

$$\begin{aligned}
\Phi\,\langle rcptWin\rangle &= 0\\
\Delta\,\langle rcptWin\rangle &= \Phi\,\langle msgOfst\rangle + \Phi\,\langle msgWindow\rangle + \Delta\,\langle msgWindow\rangle\\
&\quad + \Phi\,\langle rcptBuffer\rangle + \Delta\,\langle rcptBuffer\rangle\\
&= |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle + (|msgHdr| + |msgBdy|)\,\Delta\,\langle \mathtt{clkUnit}\rangle +\\
&\quad |rcptBuffer|\,\Delta\,\langle \mathtt{clkUnit}\rangle\\
&= (|msgOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle \mathtt{clkUnit}\rangle\\
\Phi\,\langle rcv\rangle &= \max\{\Phi\,\langle ch\rangle\,, \Phi\,\langle rcptWin\rangle\}\\
&= \max\{\Phi\,\langle \mathtt{delay}\rangle + |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\,, 0\}\\
&= \Phi\,\langle \mathtt{delay}\rangle + |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
\Delta\,\langle rcv\rangle &= \max\{0, \min\{\Phi\,\langle ch\rangle + \Delta\,\langle ch\rangle\,, \Phi\,\langle rcptWin\rangle + \Delta\,\langle rcptWin\rangle\} - \Phi\,\langle rcv\rangle\}
\end{aligned}$$

Since $\Phi\,\langle rcptWin\rangle + \Delta\,\langle rcptWin\rangle > \Phi\,\langle ch\rangle + \Delta\,\langle ch\rangle$ and $\min \Delta\,\langle rcptBuffer\rangle \geq \max \Phi\,\langle \mathtt{delay}\rangle$ (see section 5.3):

$$\begin{aligned}
\Delta\,\langle rcv\rangle &= \max\{0, \Phi\,\langle ch\rangle + \Delta\,\langle ch\rangle - \Phi\,\langle receiver\rangle\}\\
&= \Delta\,\langle ch\rangle\\
&= (|msgHdr| + |msgBdy|)\,\Delta\,\langle \mathtt{clkUnit}\rangle
\end{aligned}$$

The maximum time needed for a receiver to receive a message is:

$$\begin{aligned}
\Phi\,\langle rcv \parallel rcptWin\rangle &= \min\{\Phi\,\langle rcv\rangle\,, \Phi\,\langle rcptWin\rangle\}\\
&= \min\{\Phi\,\langle \mathtt{delay}\rangle + |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\,, |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle\\
&= |msgOfst|\,\Phi\,\langle \mathtt{clkOfst}\rangle
\end{aligned}$$

$$\begin{aligned}
\Delta\,\langle rcv \parallel rcptWin\rangle &= \max\{0, \max\{\Phi\,\langle\mathtt{delay}\rangle + |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + \\
&\quad (|msgHdr| + |msgBdy|)\,\Delta\,\langle\mathtt{clkUnit}\rangle, \\
&\quad |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle\} \\
&\quad - |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle\} \\
&= \max\{0, |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgHdr| + |msgBdy| + \\
&\quad |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle - |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle\} \\
&= (|msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle
\end{aligned}$$

This is true, because $\min\Delta\,\langle rcptBuffer\rangle \geq \max\Phi\,\langle\mathtt{delay}\rangle$.

Furthermore, TEA requires, that all controllers close the reception window in the same slot. To overcome the difference between the clocks of the slowest and fastest controller, the receivers wait for another message offset.

$$\begin{aligned}
\Phi\,\langle receiver\rangle &= \Phi\,\langle rcv \circ rcptWin\rangle \\
&= |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle \\
\Delta\,\langle receiver\rangle &= \Delta\,\langle rcv \circ rcptWin\rangle + \Phi\,\langle msgOfst\rangle \\
&= (|msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle + |msgOfst|\,\Phi\,\langle clkOfst\rangle \\
&= (|msgOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle
\end{aligned}$$

Productions 5.9 and 5.7 The action *transfer* includes the parallel actions *sender*, *neighbor* and *receiver*:

$$\begin{aligned}
\Phi\,\langle transfer\rangle &= \min\{\Phi\,\langle sender\rangle, \Phi\,\langle neighbor\rangle, \Phi\,\langle receiver\rangle\} \\
&= \min\{|msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle, |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle\} \\
&= |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle \\
\Delta\,\langle transfer\rangle &= \max\{0, \max\{\Phi\,\langle sender\rangle + \Delta\,\langle sender\rangle, \Phi\,\langle neighbor\rangle + \Delta\,\langle neighbor\rangle, \\
&\quad \Phi\,\langle receiver\rangle + \Delta\,\langle receiver\rangle\} - \Phi\,\langle transfer\rangle\} \\
&= \max\{0, \max\{|msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgHdr| + |msgBdy|)\,\Delta\,\langle\mathtt{clkUnit}\rangle, \\
&\quad |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgHdr| + |msgBdy| + |msgOfst|)\,\Delta\,\langle\mathtt{clkUnit}\rangle, \\
&\quad |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|) \\
&\quad \Delta\,\langle\mathtt{clkUnit}\rangle\} - |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle\} \\
&= \max\{0, |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + (|msgOfst| + |msgHdr| + |msgBdy| \\
&\quad + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle - |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle\} \\
&= (|msgOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle
\end{aligned}$$

All other partitions are depending on the slot which can be given, now. The duration of the slot becomes equal to its length, since the □-operator is used.

$$\begin{aligned}
\Phi\,\langle slot\rangle &= 0 \\
\Delta\,\langle slot\rangle &= \Phi\,\langle slotOfst\rangle + \Phi\,\langle transfer\rangle + \Delta\,\langle transfer\rangle \\
&= |slotOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + |msgOfst|\,\Phi\,\langle\mathtt{clkOfst}\rangle + \\
&\quad (|msgOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle \\
&= (3\,|slotOfst| + |msgHdr| + |msgBdy| + |rcptBuffer|)\,\Delta\,\langle\mathtt{clkUnit}\rangle
\end{aligned}$$

because $|slotOfst| = |msgOfst|$ (see section 5.3).

Productions 5.4, 5.5, 5.2, 5.3 and 5.1 The following views are given in terms of slots instead of clock units or clock offsets.

$$\begin{aligned}
\Phi\ \langle reqPhase\rangle &= 0\\
\Delta\ \langle reqPhase\rangle &= |reqPhase|\,\Delta\ \langle slot\rangle\\
\Phi\ \langle confirmPhase\rangle &= 0\\
\Delta\ \langle confirmPhase\rangle &= |confirmPhase|\,\Delta\ \langle slot\rangle\\
\Phi\ \langle regPart\rangle &= 0\\
\Delta\ \langle regPart\rangle &= (|reqPhase| + |confirmPhase|)\,\Delta\ \langle slot\rangle\\
\Phi\ \langle extPart\rangle &= 0\\
\Delta\ \langle extPart\rangle &= |extPart|\,\Delta\ \langle slot\rangle\\
\Phi\ \langle cycle\rangle &= 0\\
\Delta\ \langle cycle\rangle &= |cycleOfst|\,\Phi\ \langle clkOfst\rangle + (|reqPhase| + |confirmPhase| + |extPart|)\,\Delta\ \langle slot\rangle
\end{aligned}$$

9.3 Efficiency

9.3.1 Bandwidth Loss Due to Limited Clock Precision

Due to the fact, that the clocks of the controllers cannot be synchronized perfectly it is necessary to wait several times in the cycle to assure that all controllers are in the same cycle, the same slot or the same phase within a slot. This costs bandwidth, and therefore the maximum differences between two clocks should be as low as possible. *Efficiency* means, that more bandwidth can be used for data transmission, so that the waiting times within the slots is low.

Two values are important: $|cycleOfst|$ is the number of clock offsets a controller waits before starting with the first slot. If δ^0_{clock} (see section 2.6.2) is the offset after the last synchronization attempt (presumedly at the end of the cycle), then $\min |cycleOfst|\,\Delta\ \langle \mathtt{clkOfst}\rangle \geq \delta^0_{clock}$. This value depends primarily on the quality of the clock synchronization algorithm. On the other side, the evaluated grammar can be used to calculate the number of clock offsets (or clock units) $|slotOfst|$ which are necessary to overcome the difference between the slowest and the fastest clock at the end of the cycle. This means,

$$\min \Delta\ \langle \mathtt{slotOfst}\rangle \geq \max \Delta\ \langle cycle\rangle - \min \Delta\ \langle cycle\rangle$$

This approach can be used to determine the loss of bandwidth within a cycle. In the following *clkOfst* is already substituted by *clkUnit* and *msgOfst* substituted by *slotOfst*. Also, $s = |regPart| + |extPart| = |reqPhase| + |confirmPhase| + |extPart|$ should be the total number of slots in the cycle, and $m = |msgHdr| + |msgBdy|$ the total length of the message in clock units. The nominal length of a clock tick is $\hat{t}$.

$$
\begin{aligned}
& \min \Delta \ \langle \texttt{slotOfst} \rangle \geq \max \Delta \ \langle cycle \rangle - \min \Delta \ \langle cycle \rangle \\
\Rightarrow \quad & |slotOfst| \, \gamma_f \, \hat{t} \geq |cycleOfst| \, \gamma_s \, \hat{t} + s \, \max \Delta \ \langle slot \rangle - |cycleOfst| \, \gamma_f \, \hat{t} \\
& \qquad - s \, \min \Delta \ \langle slot \rangle \\
& \quad = |cycleOfst| \, (\gamma_s - \gamma_f) \, \hat{t} + s \, (\max \Delta \ \langle slot \rangle - \min \Delta \ \langle slot \rangle) \\
& \quad = |cycleOfst| \, (\gamma_s - \gamma_f) \, \hat{t} + s \, ((3 \, |slotOfst| + m + |rcptBuffer|) \, \gamma_s \, \hat{t} \\
& \qquad - (3 \, |slotOfst| + m + |rcptBuffer|) \, \gamma_f \, \hat{t}) \\
& \quad = |cycleOfst| \, (\gamma_s - \gamma_f) \, \hat{t} + s \, (3 \, |slotOfst| + m + |rcptBuffer|) \, (\gamma_s - \gamma_f) \, \hat{t} \\
& \quad = |cycleOfst| \, (\gamma_s - \gamma_f) \, \hat{t} + 3 \, s \, |slotOfst| \, (\gamma_s - \gamma_f) \, \hat{t} \\
& \qquad + s \, (m + |rcptBuffer|) \, (\gamma_s - \gamma_f) \, \hat{t}
\end{aligned}
$$

so that

$$|slotOfst| \, \gamma_f \, \hat{t} - 3 \, s \, |slotOfst| \, (\gamma_s - \gamma_f) \, \hat{t} \geq (|cycleOfst| + s(m + |rcptBuffer|)) \, (\gamma_s - \gamma_f) \, \hat{t}$$

and

$$|slotOfst|(\gamma_f - 3 \, s \, (\gamma_s - \gamma_f)) \, \hat{t} \geq (|cycleOfst| + s \, (m + |rcptBuffer|)) \, (\gamma_s - \gamma_f) \, \hat{t}$$

It is assumed, that $\gamma_f - 3 \, s \, (\gamma_s - \gamma_f) > 0$, because the definition of the grammar does not allow for a negative solution for $|slotOfst|$. Also, $|slotOfst|$ must be an integral value. Therefore, the solution becomes

$$|slotOfst| \geq \left\lceil \frac{(|cycleOfst| + s \, (m + |rcptBuffer|)) \, (\gamma_s - \gamma_f)}{\gamma_f - 3 \, s \, (\gamma_s - \gamma_f)} \right\rceil$$

with $|rcptBuffer| \geq |slotOfst|$.

9.3.2 Examples

The amount of bandwidth loss depends on three factors. These are the quality of clock synchronization ($|cycleOfst|$, $\Delta \ \langle \texttt{clkUnit} \rangle = [\gamma_f; \gamma_s]$), the number of slots and the length of the message including the time that is needed for the reception buffer. This implicitly includes the channel delay.

Figure 9.1 shows the value for $|slotOfst|$ for different numbers of slots and for different clock drifts. The number of slots varies from 10 to 60.

The parameter r is the maximum deviation of the synchronized clocks. It ranges from 10 to 200 ppm. The minimum and maximum clock speed are set symmetrically around the nominal clock frequency. This means, that $r_{min} = -r$ and $r_{max} = r$, so that $\gamma_f = 1 - r$ for the fastest clock, and $\gamma_s = 1 + r$ for the slowest clock. The cycle offset is set to 100 clock offsets.

The length of the message is based on the assumptions introduced in section 6.5. This means, that a byte is encoded with 10 bits, there is an introductory sequence of three bits and a trailing sequence of 12 bits to assure that the channel became idle. The bit-time on the channel is estimated as $0.1 \, \mu s$. This requires a 10 MBit/s transmission media. The delay is assumed to be not greater than $a \cdot 100 \, ns$. Together, the nominal message and reception buffer length in clock units are

$$100 \, (12 + 3 + 10 \cdot 256) + 100 = 257600$$

for 256 byte per message including the message header and body.

The bandwidth loss depending on the message length with fixed number of slots (15 slots in this example) is shown in figure 9.2. The different cases shown are a quarter and the half of the message size above, and the double and four times the size of the original message.

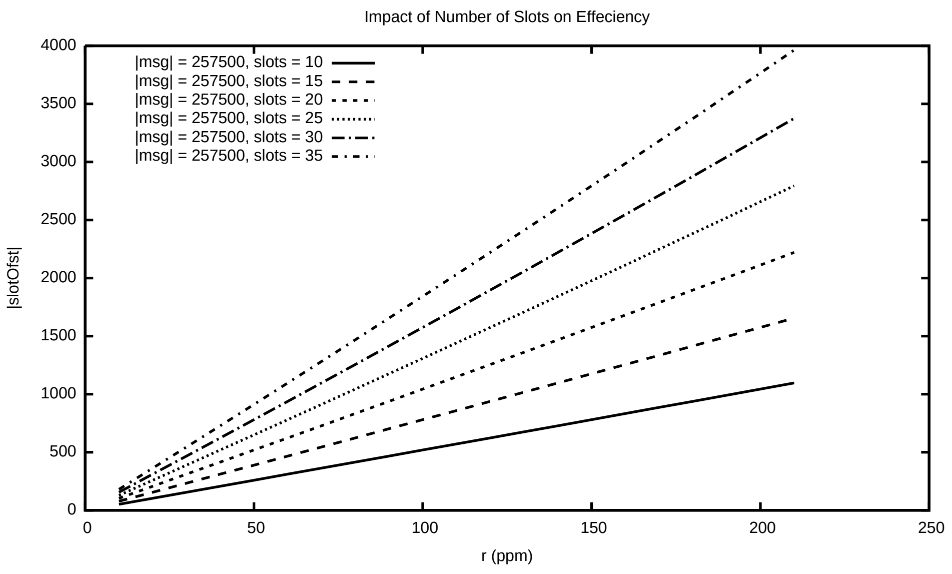


Figure 9.1: Bandwidth loss per slot depending on the number of slots and the quality of the clock.

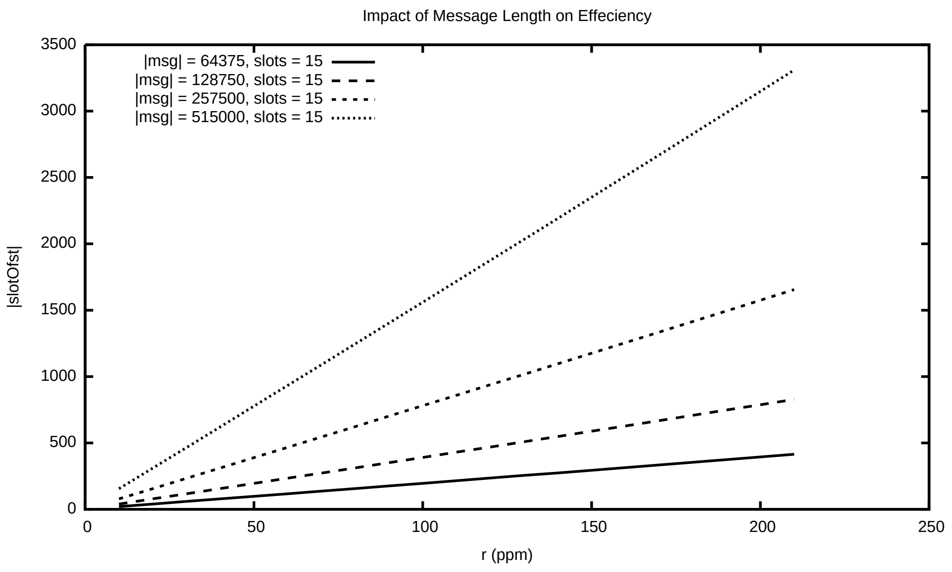


Figure 9.2: Bandwidth loss per slot depending on the message size and the quality of the clock using 15 slots.

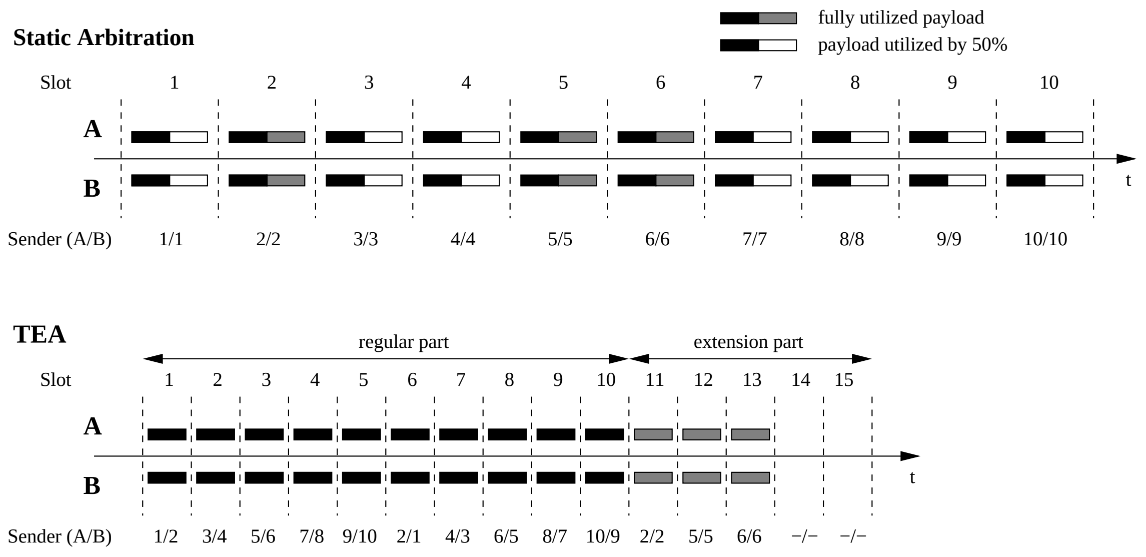


Figure 9.3: In contrast to the protocol with static slot scheme, the TEA variant uses shorter slots, but reserves additional bandwidth in the extension part. Although all messages have the same length, they do not utilize the allocated bandwidth in case of the static scheme. TEA can allocate extra bandwidth dynamically.

9.4 Case Study: Requesting Additional Bandwidth on Demand

This case study presents a scenario, where TEA can provide a more efficient usage of bandwidth than a protocol with static arbitration. It is assumed, that each sender has to send a variable amount of data, so it does not utilize the full bandwidth which is reserved for the sender. A protocol without dynamic arbitration has to reserve the bandwidth which allows to transmit the maximum possible amount of data, even if the probability for this case is very low.

If the case that all senders try to use the maximum bandwidth in the same cycle can be excluded (otherwise, it may be possible to use a sophisticated scheduling strategy – see chapter 10), it is possible in TEA to shorten the message length for all controllers while extending the cycle by an extension part, where the senders can request additional bandwidth if necessary.

Figure 9.3 shows a scenario, where the TEA protocol can make use of half the message length of the static protocol. Additionally, the TEA configuration provides five extension slots of the same length as the static slots, which can be used if more bandwidth is required.

The following assumptions were made:

- There are 10 controllers in the network, each with one slot per cycle (static scheduling) or one slot in the regular part of TEA.
- The extension part has five slots of the same length as the slots in the regular part.
- The message length is 256 byte. Also there is an encoding offset of 3 + 12 bit. The reception buffer is set to 100 clock units. Ten bits are needed to encode a byte and the bit time is $100\,ns$ on the channel.
- The message length, when using the TEA protocol, is only 128 byte. Besides the encoding delay, there is an additional overhead for the confirmation vector (see section 8.1.2).
- The (synchronized) clock offset at the start of the cycle is up to $100\,ns$.

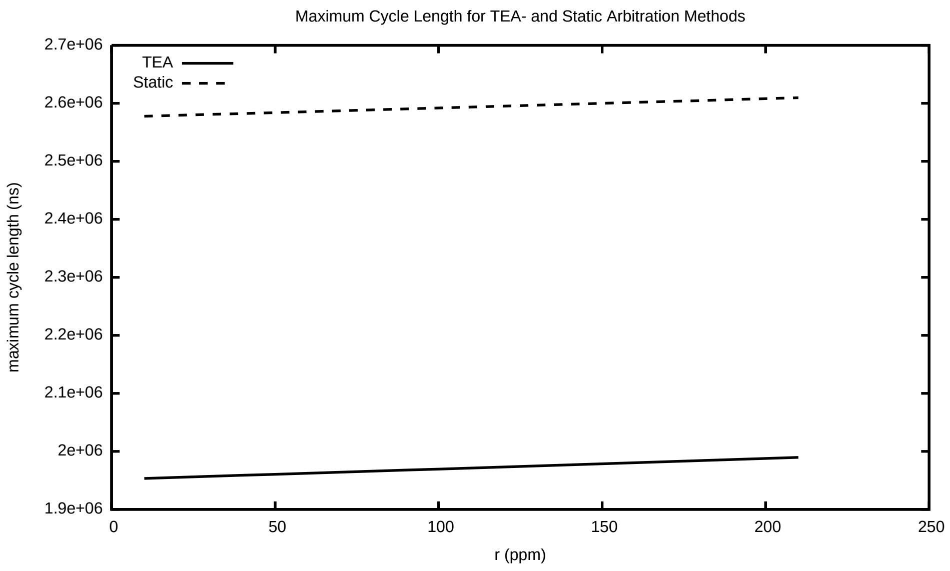


Figure 9.4: Absolute cycle length in real-time depending on the clock drift in the scenario used in the case study. In contrast to the dynamic allocation scheme in TEA, the usage of a static allocation scheme results in significantly longer cycles.

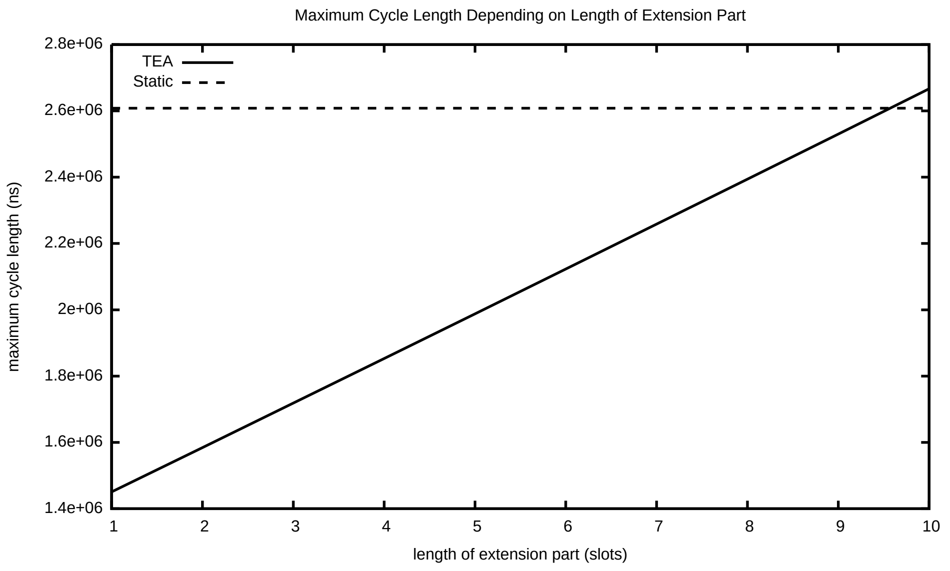


Figure 9.5: The numbers of slots in the extension part in relation to the cycle length. Up to 9 slots can be configured in the extension, before the cycle length using the dynamic allocation scheme becomes higher than when using a static allocation scheme.

- The (synchronized) clock skew ranges from 10 to 200 ppm.

Figure 9.4 shows that the cycle length which is needed when using the TEA protocol is significantly shorter than when using a static allocation scheme. This raises the question how many extension slots can be configured to achieve the same cycle length. The relationship between extension slot numbers and cycle length is shown in figure 9.5. It shows that up to 9 additional slots can be configured, before the TEA cycle length reaches the fixed cycle length when using a static allocation method. This is nearly what is expected: The number of slots is doubled, while the messages are half of the original message size.

10 Scheduling

In contrast to the regular part, the schedule in the extension part is built up dynamically. The requested slots are allocated to the requesting controllers in the extension.

All controllers have to decide consistently which controller should send in the extension. The basis of this decision of each controller is the vector *result_vector*, which is the outcome of the agreement algorithm in section 8.1.3. Therefore each controller holds the necessary information in a local register `dyna_sched`, which is built up from the final request vector *result_vector*. The `dyna_sched` register is the basic register used by the scheduling unit of the communication controller as described in section 6.4.3. If the number e of slots available in the extension is lower than the number of requesting controllers, the controllers have to be scheduled to slots in subsequent cycles.

Typically, a schedule is subject to one or more scheduling policies. This includes round-robin-, FIFO- and static priority policies and combinations of these. In the following, formal definitions will be given for these policies in the context of the TEA protocol. Section 10.2 will show possible low-cost algorithms for the implementation of these policies (see also [Lis05b]).

Two bit vectors are used for a formal definition of the policies: The element $s_{i,k}$ in vector $\mathbf{s}_k$ is set, if the result of the agreement algorithm in cycle k is *result_vector*(i) = `request`. It remains set until it is selected for sending in the extension part. It is cleared immediately after sending. The state of the vector s_k after the extension part and before cycle $k+1$ is held in the vector p_k. For each cycle, p_k is set to s_k after the extension part has ended. In cycle 0 (the first cycle), each element $s_{i,0}$ is initialized with $s_{i,0} = 1$ if *result_vector*(i) = `request` after the first regular part, and $s_{i,0} = 0$ otherwise. p_0 is set after the first extension part.

A vector $\mathbf{s}_k$ is processed by a round-robin policy, if the requesting controllers are scheduled according to their given order. After all controllers have been processed, the schedule starts anew with the first controller i in $\mathbf{s}_k$ where the bit $s_{i,k}$ has been set. This is the basic strategy as used by the controllers described in section 6.4.3. The following section defines other policies based on the basic round-robin strategy. Figure 10.1 illustrates how the policies are related to each other.

10.1 Scheduling Strategies

10.1.1 FIFO-policy

The FIFO-policy is basically built up on the round-robin policy, with the exception, that if two controllers requested slots in different cycles, the controller which has requested its slot earlier is selected first. Let $\text{age}(i,k)$ be the number of cycles controller i is waiting for a slot in cycle k. The function $\text{age} : C_t \times \mathbb{N} \mapsto \mathbb{N}$ is defined by

$$\begin{aligned} \text{age}(i,0) &= s_{i,0} \\ \text{age}(i,k) &= k - \max\{h \in \mathbb{N}^+ \mid p_{i,h-1} = 0 \wedge h \le k\} \end{aligned}$$

The following two conditions hold for the FIFO-policy:

1. If $\text{age}(i,k) > \text{age}(j,k)$, then i is selected before j.
2. If $\text{age}(i,k) = \text{age}(j,k)$, then i and j are selected according to the round-robin policy.

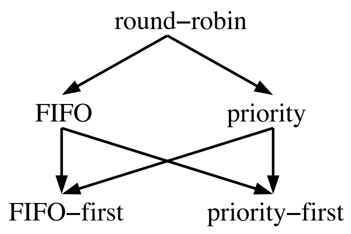


Figure 10.1: The dependencies between the policies.

10.1.2 Static Priorities

The static priority policy is based on a function prio(i) $\in \{0, \ldots, p\}$, where $p \in \mathbb{N}$ is the highest priority. The following two conditions hold for the priority policy:

1. If prio(i) > prio(j), then i is selected before j.
2. If prio(i) = prio(j), then i and j are processed according to the round-robin policy.

Note, that it is possible that a controller i with low priority is never selected.

10.1.3 Priority-First Policy

Priority and FIFO policies can be combined to two new policies, depending on the order the policies are applied.

In the priority-first policy, the set of the requesting controllers with highest priority is taken, and then the first controller is scheduled according to the FIFO-policy. The following conditions hold for the priority-first policy;

1. If prio(i) > prio(j), then i is selected before j.
2. If prio(i) = prio(j) and age(i, k) > age(j, k), then i is selected before j
3. If prio(i) = prio(j) and age(i, k) = age(j, k), then i and j are selected according to the round-robin policy

10.1.4 FIFO-First Policy

In the FIFO-first policy, the set of all i with highest age(i, k) is taken, and then scheduled according to the priority policy. The following conditions hold:

1. If age(i) > age(j), then i is selected before j.
2. If age(i) = age(j) and prio(i) > prio(j), then i is selected before j
3. If age(i) = age(j) and prio(i) = prio(j), then i and j are selected according to the round-robin policy

10.2 A Low-cost Algorithm

A scheduling logic in a controller chip requires hardware costs in term of registers. Therefore, a possible solution should require only a minimal number of registers with minimal size. In the following, the hardware costs will be given in terms of the sum of bits of all registers to be implemented in each controller. The following algorithm will only use a bit vector with one bit per controller, and two or three index registers depending on the combination of scheduling policies – in all, a very low number of hardware elements.

This section presents the algorithm, that is the basis of the various scheduling strategies mentioned in the last section. The properties of the algorithm, especially in the case of faults, are discussed section 10.3.

The implementation in hardware uses registers for accounting of requesting controllers, and two index registers. The register `dyna_sched` is a bit vector which has the same properties as the vector $\mathbf{s}_k$. This register is updated each cycle. The registers `next` and the temporary register `tmp` are used to locate bits in *dyna_sched* and are common to each implementation. The index registers have cyclic behavior.

Each implementation relies on the three subroutines `merge()`, `select()` and `adjust()`. The first one is used to merge the outcome of the agreement algorithm into the register `dyna_sched`. `select()` is the selection function which returns the controller to be activated next in the schedule and is called immediately before the respective slot. The subroutine `adjust()` is called immediately after the respective bit in register `dyna_sched` has been cleared. It adjusts the registers to fulfill the necessary constraints for the next selection. In this case, the `next` register should point to a bit in `dyna_sched` which has been set. The different policies implement these elementary functions in different ways. The used implementations of the different subroutines for the different policies roughly follows the relationship given in figure 10.1. The timing of the subroutine calls is shown in figure 10.2.

The basic algorithm is a natural implementation of the round-robin strategy (see figure 10.3). `merge()` copies the outcome of the agreement algorithm into the *dyna_sched* register where the respective bit is not set (otherwise, the bit remains set). This is done at the end of the regular part. An exceptional situation arises if the `next` index points to a cleared bit in *dyna_sched* . This can be possible after the execution of one of the scheduling algorithms in the extension part. In this case, the `merge()` subroutine must set the `next` index to the first occurrence of a request. The `tmp` register is needed for iterating over *dyna_sched* . In the extension part, the controllers call the subroutine `select()` at the beginning of every slot. It returns the current value of `next` . At the end of each slot, `advance()` is called, which clears the bit at index `next` and adjusts the register `next` to the index of the next bit set.

This simple algorithm can be easily extended to implement FIFO-behavior. For this purpose, the controllers must implement a new `merge` subroutine. Given that a controller i should be merged into the *dyna_sched* register. If the bit is already set, it is left untouched. Otherwise, the temporary index `tmp` searches decreasingly from `next` $- 1$ downwards for a set bit. If it passes the index i, then the bit at index i can be set. If another set bit is found before i, the merge is rejected. If the temporary index becomes 0 it continues its search with the index $c - 1$, because of its cyclic behavior. This implementation is illustrated in fig. 10.4.

The priority policy is implemented by a modified `select` subroutine. In the extension part, the temporary register is set to the value of the `next` register, and then advances to the next index of a set bit. If the respective controller has higher priority than the controller which is indicated by the current `next` index, the `next` index is moved to the position of the temporary index. The subroutine terminates, if the temporary register points to the same bit as the `next` index again. Note, that the order of the selected controllers for scheduling is completely independent from the time they requested the slot, as is the case in the basic round-robin algorithm.

The implementation of the FIFO-first policy re-implements the `merge()` subroutine that way, that the outcome

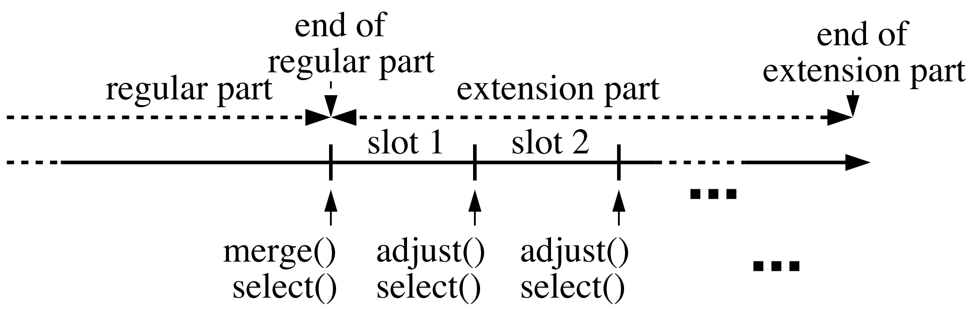


Figure 10.2: Subroutine calls in the extension part.

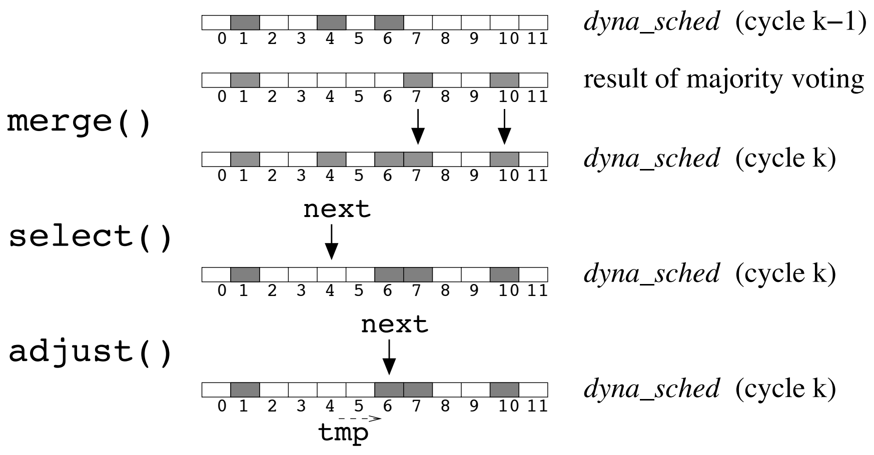


Figure 10.3: The basic round-robin algorithm. In the first step, the outcome of the agreement algorithm (as bit vector) is merged into the *dyna_sched* register. Then the controller 4 is selected and the respective bit is cleared. The index `next` must then be adjusted, so that it points to the next set bit.

of the agreement algorithm is only merged if there is no bit set in `dyna_sched`. Additionally the `select()` subroutine from the implementation of the priority policy is reused.

The priority-first implementation modifies the `merge()` subroutine from the FIFO policy. First, it is checked if the requesting controller has a higher priority than all other in `dyna_sched`. If so, it is merged. If there are entries according to controllers with higher or same priority, the algorithm follows the strategy of the `merge()` subroutine of the FIFO policy, with the exception, that the temporary register stops searching only if an entry according to a controller with the same priority is found. For the selection, the priority-first implementation also modifies the `select()` subroutine of the priority implementation. In this variant of the subroutine, the `next` register is only moved, if the bit `next` points to is cleared. Then it is moved to the next set bit, regardless of the corresponding controllers priority. Since the `next` index is not moved, the index of the bit representing a controller with highest priority must be memorized, while the temporary register iterates over the register `dyna_sched`. Therefore a third index register must be implemented.

10.3 Properties

10.3.1 Hardware Requirements

Each implementation requires at least $c + 2\lceil \log_2 c \rceil$ register bits. Since TEA needs at least 8 controllers, the minimum register space is 14 bit. The priority-first implementation requires one more index register with additional $\lceil \log_2 c \rceil$ bits. Therefore a total of 17 bit minimum are necessary for priority-first.

10.3.2 Fault-free Operation

All implementations behave according to the specification of the policies. The round-robin implementation uses the fact that the cells of the register *dyna_sched* are ordered according to the identification numbers of the controllers.

The `merge()` subroutine of the FIFO implementation assures that all newly inserted requests from the current cycle are in the next step selected after the older ones. It assures, that all requests that cannot be processed in

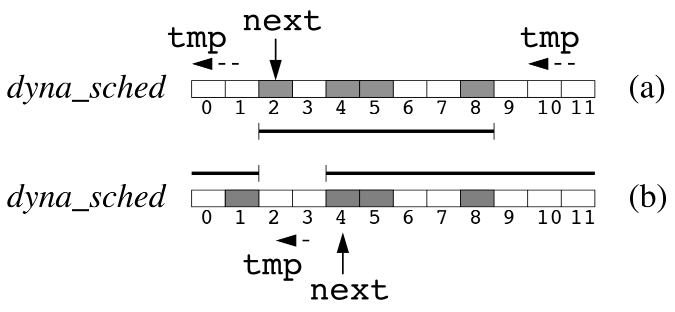


Figure 10.4: The merge subroutine in the FIFO-implementation. The black bars indicate the regions where no new request can be merged into.

FIFO-order are denied by setting only bits for controllers between the last bit set and the position of the `next` index. If a controller cannot be processed in FIFO-order it has to request in the next cycle again, until the `next` index passes its position. Then, it is the last in the FIFO-queue and this fulfills the first-in-first-out order.

In the priority policy implementation, the `next` index register is only moved if an entry for a controller with higher priority is found. If none is found, the respective controller has highest priority, and is the first in round-robin order. Fig. 10.4 shows an example. In (a) controller 1 requests a slot, and is successfully merged (b). Since the temporary index cycles backwards from the current `next` index, the index 1 is found before the set bit at index 8 following the temporary index. When controller 2 is selected and the respective bit is cleared, the `next` index advances to 4. In the next cycle, controller 2 and 3 could successfully send a request.

The FIFO-first policy assures that only controllers which have requested slots at the same time are merged. This way all requests have the same age, and are then selected according to priority policy. As a result, all older requests are processed before new requests are accepted.

In the priority-first policy implementation the condition holds that after the `merge()` subroutine each two controllers with the same priority respect FIFO order. The `select()` step returns first higher priority controllers in FIFO order, and then continues with lower priority ones respecting the age of the request, because the `next` index is not moved, until the corresponding controller has been selected. The black bars in fig. 10.5 indicate the regions in *dyna_sched* where bits should not be set depending on their priority.

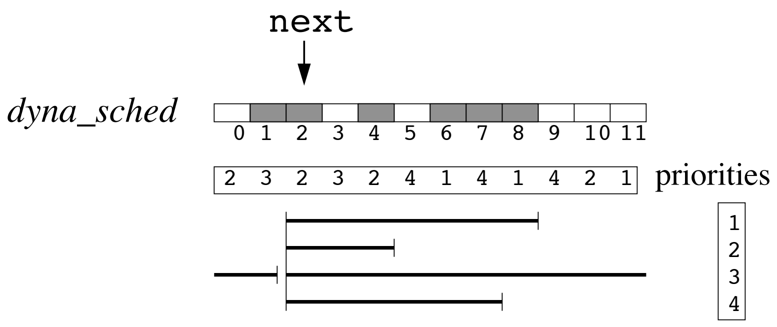


Figure 10.5: In priority first, the FIFO's with higher priorities (from bottom to top) are processed first. The index `next` points to the next cell with highest priority.

10.4 Operation in the Presence of Faults

10.4.1 Controller Faults

A faulty controller can behave in three different ways. If a controller does not request a slot, it can try to access the media or not. In any case, the neighbor controller which guards it prevents the faulty controller from sending. Note that double controller faults are excluded by the fault model. Additionally a faulty controller can prevent its neighbor from sending in its slot.

A faulty controller can also request a slot, and then access the media or not. A corrupted message or an empty slot can be safely ignored by the fault-free controllers. If this is the case, the slot can be shortened as described in [Lis05a]. Any controller continues with the next slot in the schedule. If the neighbor is scheduled in the extension, too, and the faulty controller prevents its neighbor from sending or corrupts its message by denying access to the media, the neighbors slot may be shortened, too. In this case there may be enough space for an additional slot.

It is possible that a faulty controller who requests a slot ignores FIFO order. However, the `merge()` subroutines prevent such a request at the receivers by ignoring the request while merging as described in 10.2. For example, the controllers would not set bit 7 in fig. 10.4 even if the faulty controller 7 requests a slot.

Third, the outcome of the agreement algorithm after the regular part could be `unknown_request`. The receivers must decide if `unknown_request` values count as requests or not. TEA uses a simple rule in the `merge()` subroutine: The controller will be scheduled for the extension part, if the respective bit in the `dyna_sched` register is set, regardless of what it sent in the time between the request and the time it gets its slot in the extension. In other words: A controller cannot retract a requested slot.

In case of controller faults it can happen, that slots are allocated for a faulty controller or a blocked neighbor, or a faulty controller is not scheduled for a slot. Both situations have been discussed at the beginning of this sub-section on controller faults.

10.4.2 Channel Faults

Channel faults are tolerated in the TEA protocol in the sense, that the controllers reach agreement at the end of the regular part. However, if the controller sends to the faulty channel in the request phase of the regular part, the outcome of the agreement can be the value `unknown_request`.

It is not possible to distinguish a controller fault from a channel fault, if the fault-free controllers agree in the value `unknown_request` for the faulty controller. For this reason, channel faults are handled the same way as controller faults. Without an appropriate countermeasure this may prevent up to $\frac{c}{2}$ fault-free controllers from being ever scheduled for the extension, independent from their priority. The problem is solved, if all requests are sent on the fault-free channel in the request phase. This means that all requests must be sent on both channels.

In the remainder of the section, three alternative solutions are presented. The choice of the solution depends mainly on the constraints of the application, bandwidth and hardware costs.

- Double the request phase before the confirmation phase, and reverse the assignment of the controllers to the channels in the second request phase. This means a controller sends its request to one channel in the first request phase and to the other in the second request phase. This enlarges the cycle permanently by $\frac{c}{2}$ slots. So this is mainly a solution if one suspects massive message corruption on a faulty channel.
- Reverse the assignment to the channels in the request part of the next cycle. This means a controller sends its request to one channel in all even cycles and to the other channel in all odd cycles. Requesting controllers whose messages become corrupted through channel faults in the first cycle, are delayed by one cycle.

- Add a link in each node between both controllers, so that a controller can inform its neighbor of its intent to send a request, and the controller and the neighbor distributes the request to both channels in the request phase. This requires that both neighboring controllers are scheduled for different channels in the request phase. Also, one additional bit must be reserved in the messages for the neighbors request.

Any of these solutions ensure that there are always well defined values for any controller known at the end of the regular part at least after two cycles. Therefore the scheduling in the extension is not adversely affected by channel faults.

11 Dynamic Slot Length in the Extension Part

The strategy of TEA to determine the schedule right before the start of the extension part leads to the possibility of building schedules dynamically. But channel access works in the same way in both the regular and extension part. Especially it is not possible to allocate slots of different length. This could be advantageous, if the full message payload part cannot be utilized. Minislotting protocols solve these problems, by dynamically extending of the current slot until the sender stops sending. The FlexRay protocol, for example, uses such a minislotting approach for the dynamic part, which is optional within a cycle (see also section 2.7.5).

The drawback of the minislotting solution is that it is hard to implement it in a fault-tolerant way. Therefore, TEA uses a different solution for providing slots of dynamic length in its extension part. A *timeout-based* method has been previously described and analyzed in [Lis05a]. This chapter introduces a new method. It uses the possibility to include an extra field in the header of a message providing the information about the total slot length to the receiver. This method is, therefore, called *value-based*.

11.1 Principle of Operation

In the value-based strategy, the receivers try to receive a header of a message, and then decode its length field while receiving the rest of the message. A sender sets the length field to the desired length of the slot. In the case of a faulty sender, the message may be too short or too long to fit into a slot of that length. In the latter case it lies in the responsibility of the neighbor of the sender to close the gates, so that the sender cannot access the channels too long. In contrast to the regular part, all controllers are sending simultaneously on both channels.

Since a receiver can get the slot length from the message and the schedule is already known before the extension part starts, it is possible to protect the channels. But special care has to be taken in case of faults, especially in case of double-faults. If a faulty controller is not sending on the fault-free channel, and a byzantine fault occurs on the other channel, then some receivers may not be able to determine the slot length, while others receive a valid header (and also a valid value in the slot length field) on the faulty channel. In this case, if the slot length is not available, then a default value must be chosen. Now, if the following sender has no information about the slot length, it must not send while the current sender is transmitting its message. Therefore, it is safest to assume the maximum possible slot length.

Another exceptional situation arises, if a receiver receives valid, but different headers on both channels with different length fields. It is possible that other receivers cannot receive one of both values. The neighbor of the sender may receive only one value, either the higher or lower value. If it gets the higher value, then the following sender must not choose the lower value, so that each receiver always chooses the higher value.

In the following the set of controllers is split into the receivers in the current slot, the neighbor of the current sender (which is also a receiver) and the sender. The respective sets can be built for the subsequent slot. Especially the following sender and the following neighbor are important. The controllers must find the following slot after receiving the message within the current slot. The following sender must not start sending, as long as another controller is sending, even in the case of faults.

The following sections describe the behavior of the controllers without mentioning potential weaknesses or drawbacks. It will turn out in section 11.2.2 that communication remains possible for non-faulty controllers even in case of faults.

11.1.1 Behavior of the Receivers in the Current Slot

A receiver tries to determine the slot length from the length field in the header of the last message received. If no message could be received since the end of the last message for the amount of time of the maximum slot length, message loss is assumed. If corrupted headers were received on both channel **A** and **B** the maximum slot length is assumed. Since a following sender can have more information about the slot length available, it can start to send before the end of the maximum slot length. Therefore, the receiver prepares for the reception from the sender scheduled for the following slot. If a header is seen, then the next slot starts. Otherwise, the receiver uses the value from the header available.

The following table shows how the slot length is determined by the receivers:

Value on channel **A**	*a*	*a*	*(faulty header)*	*(faulty header)*	*a*
Value on channel **B**	*a*	*(faulty header)*	*a*	*(faulty header)*	*b*
Result	*a*	*a*	*a*	*max. length*	$\max(a,b)$

In case of a contradiction (last column) the highest value is taken. If a value could be decoded from a header, the receiver expects the slot to have this length, even if there are decoding errors in the body of the message currently receiving.

11.1.2 Behavior of the Current Neighbor Controller

Basically, the current neighbor is a receiver like the other ones. But, additionally, it is responsible for the control of the gates to the channels of the current sender. The following section explains how the neighbor determines the start of the slot. If the neighbor could decode a value from a message header, then it closes the gates after the respective amount of time if there are no faults. If any faults are detected on one channel it closes the gate of the respective channel immediately. This is to prevent fake messages from a faulty sender, who can send a second message, when the receivers expect a message from the following sender, and therefore may accept a message from the faulty controller assuming it originates from the following sender. With the neighbor shutting down the gates after a message fault has been detected, a faulty sender is not able to send a second message. Section 11.2.2 will give an example for this situation in the context of a concrete failure scenario.

The following table summarizes the behavior of the neighbor of the current sender:

Value on channel **A**	*a*	*a*	*(faulty header)*	*(faulty header)*	*a*
Value on channel **B**	*a*	*(faulty header)*	*a*	*(faulty header)*	*b*
Result	*a*	*a*	*a*	0	$\min(a,b)$

11.1.3 Behavior of the Neighbor in the Subsequent Slot

Before the sender can send, it has to observe the status of the gates which are controlled by the neighbor controller, who may come to a different conclusion about the slot length in the case of faults. This scenario will be discussed in the following section.

In the fault-free case, a receiver waits for an incoming message. After the header is read, the desired slot length is taken from the message length field. The neighbor controller closes the gates after the end of the slot. The following neighbor then opens the gates of the following sender. The following neighbor decides the slot length the same way as any other receiver. The table in section 11.1.1 show, that the slot length the neighbor assumes is always greater than or equal to the slot length the current neighbor uses. Therefore, it is not possible that there are any collisions on the channel.

11.1.4 Behavior of the Sender in the Subsequent Slot

The sender scheduled for the following slot exhibits the same behavior as its neighbor in selecting the slot length. But, because neighbor and sender can come to different conclusions, it is necessary to wait until channel access is granted. Therefore, the following sender has to monitor the switches, which are controlled by the neighbor. If it comes to the conclusion that the following slot starts it waits until the gates are open, and then it sends its message to the channels. The sender stops sending on a channel, if the gate to the channel is closed prematurely.

In the case of a faulty neighbor, it is possible, that channel access is never granted. In this case, all other receivers will experience a timeout (missing header), and assume maximum slot length. The sender should not try to send after this timeout has expired. The slot will then remain empty.

Since the following sender monitors the switches, it cannot know, if the output drivers are open in the case of driver faults. But, because the sender and the driver are part of the same fault-region, a faulty driver implies a controller region fault. The same is true for the neighbor and the switches it controls. Therefore, monitoring the switches is adequate for getting the state of the gates.

11.2 Runtime Behavior of the Controllers

With the principle of operation known, the runtime behavior of the controllers in the network must be clarified. The system must be operational in the fault-free and faulty cases. Two conditions have to be fulfilled:

1. Collisions on the channels should be avoided.
2. All controllers must agree on the same current slot number.

Although it is possible to add the current slot number to the header and to identify messages from a faulty controller this way,[1] it is the goal to provide an algorithm that does not rely on an extra header field. On the other side it is always a good idea to give a receiver the opportunity to re-check its results.

11.2.1 Behavior in the Fault-Free Case

In the fault-free case, all controllers agree on the same slot length. The headers of the messages on both channels can be received by any receiver and current neighbor and both slot length fields contain the same value. Therefore, the neighbor of the following sender will open the gates at the same time when the following sender expects the start of its slot.

11.2.2 Behavior in the Case of Faults

The scheme presented in section 11.1 makes it possible to use slots of dynamic length in the fault-free case. This section investigates the case of single- and double faults. In the following scenarios it is assumed that two controllers scheduled for subsequent slots in the extension are not part of the same node. Otherwise, situations can arise which must be handled with extra care. They are discussed in section 11.3.

[1]For this to work, the same restrictions on the schedule must be applied as described in section 11.2.2.

Single-Faults

TEA allows single non-byzantine controller faults (see section 5.1) and also single byzantine channel faults. A controller fault can affect one or both channels. It can lead to corrupted messages, or contradicting values in the slot length field of the message. Furthermore, it is possible that the neighbors' messages on one or both channels can become corrupted.

If message corruption occurs on one channel, then all receivers use the value received on the respective other channel. If both channels remain silent for the time necessary to receive a message header (and in addition the maximum channel delay), or corrupted headers are received, then the current neighbor assume that the slot has ended. Note, that in this case, the beforegoing sender or neighbor was not faulty, since this would imply two controller faults in subsequent slots, which has been excluded above. So, in this case, it is known when the header is expected by all controllers. The current neighbor closes the gates of the current sender and forbids further access to the channels. The following sender and the following neighbor are expecting the slot to have maximum length. This means, that all receivers have to wait the same amount of time before the following message must arrive.

As assumed above, the following sender is not part of the same node as the previous sender, so that it must be fault-free. The neighbor opens the gate after the maximum slot length passed, the sender observes that the gates are now open and can send the following message.

If a value fault occurs, which means that there are contradicting values in the headers, then all controllers see the same values. Since the following sender and following neighbor take the maximum value, the following sender does not access the channels before all receivers stopped to accept further signals from the current sender.

A faulty channel can behave byzantine, so that some controllers may receive a valid header, while others don't. In any case, each controller receivers a correct header on the fault-free channel. Since there are no faulty senders in this case, there are no contradictions possible, so that it is assured that each controller receives the same valid slot length field.

Double-Faults

In the case of double-faults there are two non-trivial situations possible. Other cases are already covered by single-fault or fault-free behavior.

The critical scenarios are:

1. A faulty channel causing byzantine faults, and a faulty controller causing message loss on the fault-free channel. In this case, the receivers (including the following sender and following neighbor) are split into two sets. The first one is the set of all receivers which receive no signal. The second set includes all receivers which receive a valid header with a certain value in the slot length field on the faulty channel (see figure 11.1).
2. A faulty channel causing byzantine faults, and a faulty controller sending contradicting values. In this case there are also two sets. One comprises all receivers seeing contradicting values. The receivers in the other set see one valid header on the fault-free channel (see figure 11.2).

Case 1 shows the following behavior with the rules described in section 11.1: All controllers receiving no valid header neither on channel **A** nor on channel **B** set the slot length to the maximum possible value, which means, that they are waiting for a valid message from the controller sending in the subsequent slot. On the other side, all controllers receiving a valid header on the faulty channel are setting the slot length to the requested slot length.

Since all possible behaviors of the faulty controller include the possible behaviors of the neighbor of the faulty controller, it is sufficient to investigate the case where the faulty controller is the current sender. In this case, if the neighbor receives no valid headers it closes the gates and denies the controller further access to the channels. Then, every controller sees channel faults on both channels. On the other side, if the neighbor sees a valid header on the

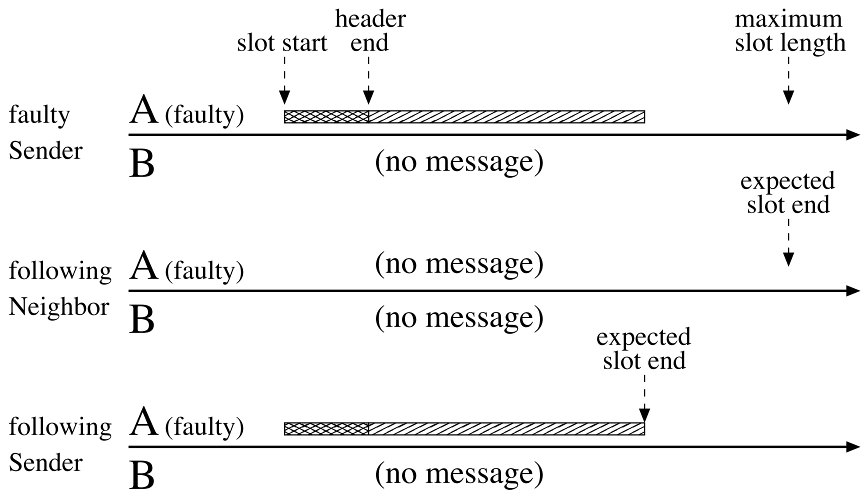


Figure 11.1: A faulty controller does not send on the fault-free channel. The following neighbor receives no message on the faulty channel. Both, the following neighbor and the following sender. have different views on the slot length.

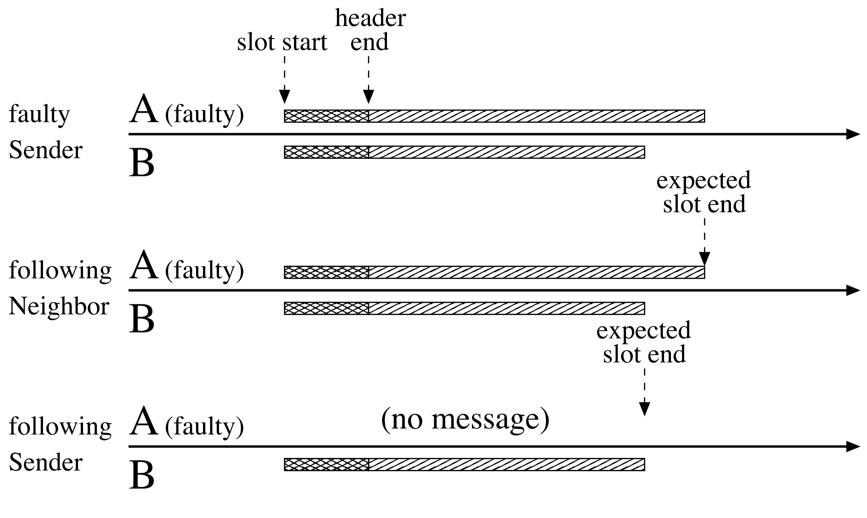


Figure 11.2: The faulty sender sends messages of different length (different values in the slot length field). The following sender receives only the shorter one.

faulty channel, then it assumes a slot of the length given in the header. In both cases, some controller will expect a specific slot length, and then wait for the header of the message from the following sender, or they will also wait for the following header for at most the maximum slot length. Since the following sender is fault-free and is not the neighbor of the faulty controller (as supposed above), then the following header is visible to all controllers on at least one channel provided that the neighbor of the following sender opens the gates before the following sender starts to send. But this requirement is always fulfilled: The sender monitors its gates and sends only if they are open and the assumed slot length has expired as described in section 11.1.4.

To show that there are no collisions, it must be clarified if the following neighbor does not open the gates before the current sender has stopped sending. Since the current sender is considered to be faulty, we can conclude, that the current neighbor must have already closed the gates before the following neighbor opens the gates of the following sender.

As supposed, two controllers scheduled for subsequent slots are not on the same node, so that each controller expected the header at the same time, because either the faulty sender is the first in the extension part, or all controllers saw the correct header on the fault-free channel from the predecessor of the faulty controller. A comparison of the behavior of the current neighbor (see table in section 11.1.1) and the behavior of the following neighbor (see table in section 11.1.3) reveals, that the current neighbor closes the gates of the current sender always before the neighbor of the sender in the subsequent slot opens the gates.

For correct operation, the condition that all controllers have the same slot counter value, must hold. The fact, that the following sender is fault-free assures this in most cases, because either the faulty message is visible, or the controller is waiting for the message from the following sender and the current neighbor does not allow channel access of the current sender.

There is one non-trivial case: A faulty controller may send a message while a receiver is waiting for a message from the following sender (for example, if a faulty header has been received). Then, the receiver accepts the header of the message from the faulty controller as the start of a message from the following (fault-free) sender.

As mentioned in section 11.1.2 the neighbor closes the gate on one channel immediately, if a fault is detected. If, the faulty sender sent a corrupted header on the fault-free channel, then this is visible for the neighbor also, so the sender has no channel access, so that the scenario described above is not possible. On the other side, if the neighbor receives a correct header on the faulty channel, then it is possible that the faulty controller has access to the channel, although another controller cannot see a valid signal (because the channel can behave byzantine). It is possible that this controller starts receiving signals at a later time in the slot. But this cannot be a valid header, because the encoding should assure, that an ongoing transmission cannot be interpreted as a valid header or sooner or later leads to a decoding fault. Because two faults cannot occur in subsequent slots it is clear, that this signal does not origin from the following sender. The case, that the neighbor does not receive a valid signal results in closing the gate on that channel, so that the sender does not have access on that channel.

As a result, the slot counters of the receivers are adjusted with the appearance of the message from the following sender which is fault-free.

The case 2 is a variation of 1. There are no slots of unknown length, but some controllers assume a higher slot length than others. This could become a problem, if the following neighbor opens the gates before the current neighbor closes the gates controlled by itself. This means, that if the current neighbor and the following neighbor disagree on the slots' length, then the current neighbor must always assume a shorter slot than the following neighbor. But this is always the case, because the current neighbor takes always the minimum value and the following neighbor always the maximum value for the current slot.

The case, that one of the controllers does not receive a valid message on one channel leads to the same conclusion, because both controllers receive at least one message on the fault-free channel.

The conclusion is that TEA tolerates both cases discussed above, and therefore all double-faults.

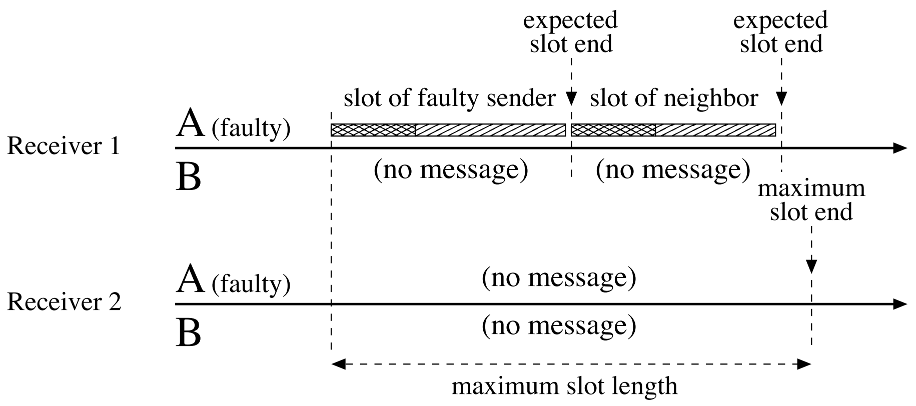


Figure 11.3: A faulty controller and its neighbor are sending in two subsequent slots. Receiver 1 obtains both messages while receiver 2 keeps waiting for the header of the neighbor. If the following controller, after the neighbor sends, then receiver 2 will assume, that this message origins from the neighbor.

11.3 Handling of Special Cases

Two questions remain unanswered:

1. What happens, if a controller runs out of space in the extension part? A controller can be scheduled for the extension, but the remaining time in the cycle may be too short for the message to be sent completely.
2. What can be done about the case when two controllers which are part of the same node are scheduled for subsequent slots? A faulty controller can cause faults in the slot of its neighbor. Then, scenarios are possible where different receivers disagree on the current slot counter.

The first question is simple to answer: The last sender knows the length of the message it is going to send. So, it can decide not to send its message. Then, the following neighbor will assume maximum slot length, and the following sender will not be able to send in this cycle. As an alternative, the controller can send a header with an invalid slot length field (the value 0 for example makes no sense). This allows a controller to announce, that the respective bit in the *dyna_sched* register should not become unset, so it remains the first sender in the following slot, depending on the scheduling strategy. This can become the default policy, if the remaining time in the cycle is clearly too short for a header.

The second question needs more attention. If two subsequent senders are part of the same node, a faulty controller sends first and then controls the channel access of the fault-free neighbor or the other way round.

Figure 11.3 shows a critical scenario: The faulty controller sends first. It fails to send on the fault-free channel, but sends on the faulty one a short message. The following sender is the neighbor of the faulty controller. But the faulty controller closes the gates while the neighbor keeps sending. If, additionally, the faulty channel causes byzantine faults it can happen, that some controllers don't receive any signal, expecting the slot to have maximum length. Other controllers receive two short and correct messages. The sum of the length of the short slots may be shorter than the maximum slot length. So there are two different sets of controllers: While some of the receivers are waiting for a slot of maximum length to pass, others have already passed two subsequent slots.

One countermeasure against such a situation is to simply forbid two controllers on the same node to become senders in subsequent slots in the extension part. This requires a change in the scheduling policy.

If this is not wanted, then it is possible to force the first sender to request slots with a pre-determined static length which is known by every controller. It allows all controllers to rely on the slot length of the first sender, without the need to determine the length dynamically. Every controller knows when to start the following slot. This rule applies only, if two controllers from the same node are sending in subsequent slots. In any other case, the length of the slot remains dynamic.

11.4 Summary

This section shows, that slots of dynamic length are possible in a fault-tolerant way with dynamic arbitration. This algorithm has the following properties:

1. Before the slot of a sender starts, which is itself faulty or the neighbor of a faulty controller, every controller knows when to start the slot. This is true, because the length of the previous slot is known in any case, because it is either
   - The first slot in the extension part.
   - The slot after the slot belonging to a fault-free sender. Then every controller sees the correct message on the fault-free channel.
   - The slot following the neighbor's slot, if allowed. Then, the slot has maximum length.
2. Collisions cannot occur, because the current neighbor closes the gates always before the following neighbor opens the gates for the following sender.
3. The following sender and the following neighbor are fault-free, because they reside not on the same node as the faulty controller.
4. The (fault-free) following sender can always send after the (fault-free) following controller opens its gates, because it monitors the gates which are opened at the latest after the maximum slot length passed.
5. All fault-free receivers can re-synchronize on the current schedule using the header of the following sender, if they were not able to determine the length of the slot by themselves.

The behavior of the controller in a slot with a faulty sender or faulty neighbor and a faulty channel (double-fault) was modeled[2] and validated by state-space exploration using the UPPAAL tool (see [Upp], [DY00] and [Pet99]) in [BS07].

[2]The model covers a superset of all possible faults in TEA.

12 Summary and Conclusions

Although, the presence of real-time networks in safety-critical environments increases, dynamic and fault-tolerant solutions in distributed environments remain a challenge. TEA is a complex solution which allows dynamic arbitration with a double-fault hypothesis. The core protocol with some extensions and the behavior in the fault-free and faulty cases have been described in this thesis.

TEA is a time-triggered protocol, using a strict hierarchical timing structure. The basic protocol uses a strict slotting scheme. Each cycle is split into two halves. The schedule of the regular part never changes during runtime. In contrast to most protocols, some slots must be shared between two controllers. Each controller which participates in dynamic arbitration, must send at least once in the first half of the regular part and at least once in the second half of the regular part, but on the other channel than it did in the first half. The controllers are sending requests for slots in the extension part, and afterwards request vectors containing all requests obtained from other controllers.

This scheme enables TEA to determine the schedule for the extension part using an agreement algorithm. The controllers can request a slot on demand. It could be proven, that this agreement algorithm works correctly. In consequence, this means every controller knows the correct schedule before the extension part starts.

With the current schedule known, it is possible for a controller to protect the channels against a faulty neighbor controller. A double-controller architecture has been selected to allow a distributed architecture with passive channels. Each controller has the ability to switch the output driver of its neighbor on and off. The switches belong to the neighbor controller, while the output driver shares one fault-region with the sender. It could be shown, that this arrangement and the usage of a double-channel architecture is able to tolerate double-faults. Faults on one node do not disturb the communication of other nodes.

Faulty behavior of a controller and/or channel can be traced by using network behavior functions, which were introduced to model the propagation of faults in the TEA network. They answer the question, if one or more faults affect other components. Network behavior functions solve this problem from an abstract point of view. Basically, a fault is described as a recursive function at an edge in a directed graph. The fault state at each edge can be different at different times. Each node in the graph represents a component. The result of a network behavior function at an incoming edge of one node can be used as input for a function at an output edge with a delay. In TEA, most components can cause delays. Another advantage of this approach is, that they are simple enough to allow automated evaluation.

The correctness of a time-triggered protocol also includes correct timing. Although, there are many well-known clock synchronization algorithms, clock skew and offset remains a general problem in time-triggered protocols, because clocks are never synchronized perfectly. This thesis shows a method to determine the correct timings from basic parameters using timed grammars.

A timed grammar provides two different views on the temporal behavior of a system. First, it describes the structure beginning from the basic time units, up to the most abstract view on the timing of the system, covering the whole runtime. This allows a good compromise between complexity and accuracy especially for time-triggered protocols with their strict hierarchical timings. On the other side, they specify how to derive the timing of each action, represented as non-terminal in a timed grammar, from the basic timing parameters. This can also be automated, so that this is a first step to provide a tool which allows parameter calculations without building mathematical models by hand.

Timed grammars take also into account, that there are always timing differences in distributed system, due to clock jitter, different delays of components and other reasons. Since TEA is modeled as a timed grammar, it is possible to derive the time, a controller has to wait before sending after the clock starts, which is an important parameter to be set at configuration time, without the need of deriving a mathematical formula. The formula is already encoded in the timed grammar, although the grammar describes only the timing hierarchy.

The problems of fault-tolerant dynamic scheduling and slots of dynamic length have been discussed. The most important scheduling strategies can be implemented with low hardware effort in a fault-tolerant way as an extension to TEA. Similarly, it could be shown, that it is also possible to implement a fault-tolerant bus arbitration strategy with TEA.

This work demonstrated, that there are successful approaches to achieve fault-tolerant dynamic arbitration. The strategy to build a schedule before the dynamic part starts prevents the protocol from the need to observe the channel for deciding which controller will access the channel next. This is a basic requirement for effective channel protection. Additional services, like slots of dynamic length and dynamic scheduling strategies, can be built reliably on top of the protocol and provide further flexibility while the protocol remains fault-tolerant.

TEA is the first real-time protocol that is time-triggered, dynamic, efficient and tolerates double-faults at the same time.

13 Further Work

In this chapter, possible directions for further development and research are identified. Although this work touches many aspects of real-time protocols, only a rough overview may be provided. In general, further research falls into the following categories:

1. Adding necessary mechanisms beyond normal operation.
2. Extending the fault-tolerance capabilities of the protocol.
3. Improve methods for analysis.

This thesis focuses on the normal operation mode, basically on arbitration methods. Similar investigations are left open for other stages of the protocol at runtime. In the startup phase the minimum number of senders is not reached, unless it is possible to startup and synchronize eight controllers at the same time. On the other side, there is a transition from the safe, fault-tolerant operation to an unsafe operation mode during shutdown. For example, a controller initiates a shutdown and a byzantine fault occurs, the minimum number of controllers cannot be guaranteed in subsequent cycles. While startup and shutdown are difficult with a single-fault assumption anyway, it is questionable if a fault-tolerant and efficient solution can be found with a double-fault assumption. On the other side, it can be assumed that methods which allow reliable fault-detection are possible.

TEA was invented with a byzantine channel fault and a non-byzantine controller fault in mind. The next step is to broaden the fault-assumption by considering byzantine controller faults, also. This improvement can lead to different directions.

- The agreement algorithm can be changed to use more cycles for information exchange.
- A hardware solution may assure, that such a situation cannot arise.

The consequence of improving the agreement algorithm is that a decision cannot be made within one regular part. A possible hardware solution may be able to transform a signal at the output of a node in a form which cannot lead to a byzantine fault. If the receivers cannot decide if the signal level is HIGH or LOW, then a device can set the signal to a defined level. This eliminates the fault, if the device is built near the channel, so that they share the same fault region. A similar solution may be found, if the value of a bit cannot be decided, because the next edge of the signal appears too early. A device must hold the signal level for a while, until it is assured, that all receivers will see the same signal. These are two possibilities how the problem of byzantine controller faults can be solved.

The third direction is the research on the basic methods used. Timed grammars are limited, because they require a strict hierarchy. A way to a lesser strict model must be found to come to a more general method. Also, timed grammars lack a notion of locality. This means, that the timing of an action is described very precisely, but there are no informations about the place where the action is performed. On the other side, the network behavior functions describe the behavior of components, but cannot express the state of a connection over a period of time. It is possible to get rid of these limitations by merging both methods. Chapter 7 shows a first approach. The results of the network behavior function can be regarded over a period of time. On the other way round, it may be possible to come to more accurate models of a network by assigning the actions to different locations.

Apart from the research directions discussed so far, an application in a real-world environment should be considered. This thesis makes use of software simulation tools and abstract models. But for a protocol to be useful it must be implemented in hardware. This means not only the implementation of the parts described in this work, but also the selection of the right methods for decoding and clock synchronization.

A The TEA Grammar

Below is the complete grammar for the basic TEA protocol as presented and discussed in chapter 5.

$$\mathit{cycle} \rightarrow \square\,(\mathit{cycleOfst}\,.\,\mathit{regPart}\,.\,\mathit{extPart}) \tag{5.1}$$

$$\mathit{regPart} \rightarrow \mathit{reqPhase}\,.\,\mathit{confirmPhase} \tag{5.2}$$

$$\mathit{extPart} \rightarrow \mathit{slot}\,.\,\ldots\,.\,\mathit{slot} \tag{5.3}$$

$$\mathit{reqPhase} \rightarrow \mathit{slot}\,.\,\ldots\,.\,\mathit{slot} \tag{5.4}$$

$$\mathit{confirmPhase} \rightarrow \mathit{slot}\,.\,\ldots\,.\,\mathit{slot} \tag{5.5}$$

$$\mathit{cycleOfst} \rightarrow \texttt{clkOfst}\,.\,\ldots\,.\,\texttt{clkOfst} \tag{5.6}$$

$$\mathit{slot} \rightarrow \square\,(\mathit{slotOfst}\,.\,\mathit{transfer}) \tag{5.7}$$

$$\mathit{slotOfst} \rightarrow \texttt{clkOfst}\,.\,\ldots\,.\,\texttt{clkOfst} \tag{5.8}$$

$$\mathit{transfer} \rightarrow \mathit{sender} \parallel \mathit{neighbor} \parallel \mathit{receiver} \tag{5.9}$$

$$\mathit{sender} \rightarrow \mathit{msgOfst}\,.\,\mathit{msg} \tag{5.10}$$

$$\mathit{msgOfst} \rightarrow \texttt{clkOfst}\,.\,\ldots\,.\,\texttt{clkOfst} \tag{5.11}$$

$$\mathit{msg} \rightarrow \mathit{msgHdr}\,.\,\mathit{msgBdy} \tag{5.12}$$

$$\mathit{msgHdr} \rightarrow \texttt{clkUnit}\,.\,\ldots\,.\,\texttt{clkUnit} \tag{5.13}$$

$$\mathit{msgBdy} \rightarrow \texttt{clkUnit}\,.\,\ldots\,.\,\texttt{clkUnit} \tag{5.14}$$

$$\mathit{neighbor} \rightarrow \square\,(\mathit{msgOfst}\,.\,\mathit{msgWindow}\,.\,\mathit{msgOfst}) \tag{5.15}$$

$$\mathit{msgWindow} \rightarrow \mathit{msg} \tag{5.16}$$

$$\mathit{gate} \rightarrow \mathit{sender} \circ \mathit{neighbor} \tag{5.17}$$

$$\mathit{ch} \rightarrow \texttt{delay}\,.\,\mathit{gate} \tag{5.18}$$

$$\mathit{rcv} \rightarrow \mathit{ch} \circ \mathit{rcptWin} \tag{5.19}$$

$$\mathit{rcptWin} \rightarrow \square\,(\mathit{msgOfst}\,.\,\mathit{msgWindow}\,.\,\mathit{rcptBuffer}) \tag{5.20}$$

$$\mathit{rcptBuffer} \rightarrow \texttt{clkUnit}\,.\,\ldots\,.\,\texttt{clkUnit} \tag{5.21}$$

$$\mathit{receiver} \rightarrow (\mathit{rcv} \parallel \mathit{rcptWin})\,.\,\mathit{msgOfst} \tag{5.22}$$

B Symbols and Abbreviations

$\|a\|$	Number of symbols in a sequence on the right side of a production for symbol a	40
$a \,.\, b$	Sequence of actions a and b	41
$a \parallel b$	Concurrent actions a and b	42
$a \circ b$	Actions b is limit of action a	44
$\Box\, a$	Subsumption of action a	45
$\langle a \rangle$	View on an action a	36
$\langle\langle a \rangle\rangle$	Entity of an action a	36
$P_v \lhd P_w$	P_v depends directly on P_w	38
$P_v \lhd^* P_w$	P_v depends on P_w	38
α	The activity function of a component network.	26
γ_i	Gradient ("speed") of the clock of controller i: $\gamma_i = 1 + r_i$	12
$\Delta\ \langle a \rangle$	Duration of a view of an action a	35
$\delta^0_{clock,i}$	Local offset from the nominal start time at the beginning of time accounting.	17
δ^0_{corr}	The correction value used for clock synchronization.	17
δ^c_{max}	Maximum distance between fastest and slowest clock in real-time at the end of a cycle.	13
$\delta^{\mathbf{ch}}_{max}$	Maximum delay of channel **ch**.	65
$\delta^{\mathbf{ch}}_{min}$	Minimum delay of channel **ch**.	65
δ^g_{corr}	Correction value for phase correction.	17
θ	Acceptance function of an input connection of a component.	31
κ	Component behavior at an output connection of the component.	26
κ', κ''	Component behavior. Certain cases are regarded as non-critical.	27
$\Lambda\ \langle a \rangle$	Length of a view of an action a	36
σ	Protocol of a component network.	25
$\Upsilon_{k,t}$	Number of outputs showing faulty behavior per total number of outputs.	27
ϕ	Network behavior of a component at the output connections.	28
$\Phi\ \langle a \rangle$	Offset of a view of an action a	35
$\mathbf{A}$	First channel of a double-channel network.	62
$a_0(v), a_1(v)$	The input thresholds used by the acceptance function of a connection v.	31
`adjust`()	Subroutine in the scheduling algorithm. Adjusts the registers so, that the next sender can be selected.	109
age(i, k)	Number of cycles controller i waits for a slot in cycle k.	107
ARINC	Aeronautical Radio Incorporated	18
$\mathbf{B}$	Second channel of a double-channel network.	62
C_h	Set of channels **A** and **B**.	62
C_t	Set of controllers in a TEA network.	62
CAN	Controller Area Network	17
ch	Message transfer over channel (timed grammar action).	57
$Chan_{\mathbf{c}}$	Channel compound **c**.	64
$clkDiff_{max}$	Maximum time a controller has to wait until the gates can be opened.	68
`clkOfst`	Clock offset (timed grammar action).	37
`clkUnit`	Clock unit (timed grammar action).	37

$confirmPhase$	Confirmation phase (timed grammar action).	55
$confirm\_vector$	Confirmation vector, a controller sends in the confirmation phase.	88
correct(b)	Correctness of bit b.	83
CRC	Cyclic Redundancy Check	11
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance	8
$Ctrl_i$	Controller compound i.	64
$cycle$	Protocol cycle action (timed grammar action).	55
$cycleOfst$	Initial offset of the cycle (timed grammar action).	55
$\mathcal{D}_g$	Set of drivers of the gates.	62
$\mathcal{D}_r$	Set of input drivers.	62
`delay`	The channel delay (timed grammar action).	57
$delay_{out}$	Delay of the output driver.	79
$drDelay_{max}$	Maximum delay of the input drivers.	66
$drDelay_{min}$	Minimum delay of the input drivers.	66
`dyna_sched`	Dynamic schedule register of a controller	67
$extPart$	Extension part of the TEA protocol (timed grammar action).	55
$\mathcal{F}_{k,t}$	Set of output connections of k, where k shows faulty behavior at time t.	26
FPGA	Field Programmable Gate Array	72
FTM	Fault-tolerant Midpoint Algorithm	16
$\mathcal{G}_t$	Set of gates.	62
$gate$	The message is passing the gate (timed grammar action).	57
$Guard_{\mathbf{c},i,j}$	Control connection from scheduler of controller j to gates of controller i on channel $\mathbf{c}$.	63
$\mathcal{I}_k$	Set of input connections of component k.	23
$\mathcal{K}$	Set of components in a component network	23
$\mathcal{K}_c$	Set of components in component compound c	24
$listen_{end}(s)$	Time when a receiver stops listening for incoming messages in slot s.	78
$listen_{start}(s)$	Time when a receiver starts to listen for incoming messages in slot s.	78
$majority(i, \mathbf{c})$	Majority vote for controller i on channel $\mathbf{c}$.	90
`merge`()	Subroutine in the scheduling algorithm. Merges the outcome of the agreement algorithm into the `dyna_sched` register.	109
$mess_{i,k}$	k^{th} bit of message sent by controller i.	83
msg	Message (timed grammar action).	56
$msgBdy$	Message body (timed grammar action).	56
$msgHdr$	Message header (timed grammar action).	56
$msgOfst$	Message offset (timed grammar action).	56
$msgWindow$	Message window (timed grammar action).	57
$mw_{end}(s)$	End of the message window in slot s.	79
$mw_{start}(s)$	Start of the message window in slot s.	79
N	Component network	23
N_c	Component compound c in network N	24
$neighbor$	Neighbor (timed grammar action).	57
`neighbor`	Function determining the neighbor of a controller.	58
`next`	Register used to locate the next sender in the `dyna_sched` register.	109
$ng_{end}(s)$	Time when the neighbor of the sender in slot s closes the gate.	79
$ng_{start}(s)$	Time when the neighbor of the sender in slot s opens the gate.	79
$Node_{i,j}$	Node containing controller compound $Ctrl_i$ and $Ctrl_j$	64
`non_request`	Value of the request bit, when the controller does not request a slot.	86
$\mathcal{O}_k$	Set of output connections of component k.	23
P_v	Shorthand notation for $part(P, v)$.	37
$part(P, v)$	Partition of a context free grammar where P is the set of productions, and v a non-terminal symbol	37

prio(i)	Priority of controller i.	108
$Pwr_{\mathbf{c},i}$	Connection between switch and output driver of gate of controller i	63
r_i	Deviation (in ppm, for example) from the nominal clock speed.	12
r_{max}	Minimum deviation of all clocks in a system.	36
$r_{max,i}$	Minimum deviation of clock i in a system.	36
r_{min}	Minimum deviation of all clocks in a system.	36
$r_{min,i}$	Minimum deviation of clock i in a system.	36
$rcptBuffer$	Reception buffer (timed grammar action).	58
$rcptWin$	Reception window (timed grammar action).	58
rcv	Receiving in the reception window (timed grammar action).	58
$rcvMsgEnd_{\mathbf{c},i}$	End of an incoming message.	69
$rcvMsgStart_{\mathbf{c},i}$	Start of an incoming message.	69
$receiver$	Receiver (timed grammar action).	58
$regPart$	Regular part of the TEA protocol (timed grammar action).	55
$reqPhase$	Request phase (timed grammar action).	55
`request`	Value of the request bit, when the controller requests a slot.	86
$request\_matrix$	Request matrix built in the confirmation phase.	86
RES(k,t)	Resulting behavior at component k at time t.	30
$result\_vector$	Result vector of the agreement algorithm.	87
$Rx_{\mathbf{c},i}$	Connection from driver on channel $\mathbf{c}$ to controller i	63
$RxCh_{\mathbf{c},i}$	Connection from channel $\mathbf{c}$ to driver of controller i	63
$\mathcal{S}_g$	Set of switches of the gates.	62
`sched`	Static schedule register of a controller.	67
`select`()	Subroutine in the scheduling algorithm. Selects the next sender in the `dyna_sched` register.	109
$sender$	Sender (timed grammar action).	56
`sender`(s, $\mathbf{c}$)	Current sender on channel $\mathbf{c}$ in slot s	58
$slot$	Slot (timed grammar action).	56
$slotOfset$	Slot offset (timed grammar action).	56
$\hat{t}$	Basic real-time unit used for clock units.	37
$T_i(c)$	Time in real-time, when the local clock of controller i reaches local time c.	12
TDMA	Time Division Multiple Access	9
TG	Timed Grammar	39
`tmp`	Register used to locate the next sender in the `dyna_sched` register.	109
$transfer$	Message transfer (timed grammar action).	56
TTP	Time-Triggered Protocol	18
$Tx_{\mathbf{c},i}$	Connection from controller i to driver on channel $\mathbf{c}$	63
$TxCh_{\mathbf{c},i}$	Connection from driver of controller i to channel $\mathbf{c}$	63
`unavailable`	Value in the request matrix, if a confirmation value is not available.	86
`unknown_request`	Value of an entry in the confirmation vector, if the value of the corresponding request bit could not be determined.	86
$\mathcal{V}$	Set of connections in a component network	23
$\mathcal{V}_c$	Set of connections in component compound c	24
$\mathcal{V}_{in,c}$	Input interface of compound c	24
$\mathcal{V}_{out,c}$	Output interface of compound c	24
valid($b, r, s, \mathbf{c}$)	Validity of bit b in a message sent by the sender in slot s on channel $\mathbf{c}$ received by a controller r.	83

Bibliography

[Aer93] Aeronautical Radio, Inc. *ARINC Specification 659: Backplane Data Bus*, Dec 1993.

[AG97] Neil C. Audsley and Allen Grigg. Timing analysis for the ARINC 629 databus for real-time applications. In *Proceedings of the 3rd IFAC Symposium on Intelligent Autonomous Vehicles, Microprocessors and Microsystems, Vol. 21*, pages 55–61. Elsevier Science B.V., 1997.

[BBE$^+$01] Ralf Belschner, Josef Berwanger, Christian Ebner, et al. FlexRay - The communication system for advanced automotive control systems. *SAE Technical Papers*, Jan. 2001.

[BFJ$^+$00] Günther Bauer, Thomas Frenning, Anna-Karin Jonsson, Hermann Kopetz, and Christopher Temple. A centralized approach for avoiding the babbling-idiot failure in the time-triggered architecture. Jun. 2000.

[BHE06] Marcel Busse, Thomas Haenselmann, and Wolfgang Effelsberg. The impact of forward error correction on wireless sensor network performance. Technical report, Department for Mathematics and Computer Science, University of Mannheim, 2006.

[Bos91] Bosch. CAN Specification Version 2.0, September 1991.

[BS07] Christian Belka and Matthias Springer. Fallstudien der Fehlertoleranz - Analyse und Entwicklung eines geeigneten Uppaal Modells zur Verifikation des Fehlerverhaltens des TDMA basierten TEA-Protokoll. Technical report, University of Duisburg-Essen, Jan. 2007.

[Buc03] Len Buckwalter, editor. *Avionics Databuses*. Avionics Communications Inc., second edition edition, 2003.

[But92] Ricky W. Butler. The SURE approach to reliability analysis. In *IEEE Transactions on Reliability*, volume 41(2), pages 210–218, June 1992.

[CES03] Vilgot Claesson, Cecilia Ekelin, and Neeraj Suri. The event-triggered and time-triggered medium-access methods. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, page 131, Washington, DC, USA, 2003. IEEE Computer Society.

[CGK$^+$01] Edmund Clarke, David Garlan, Bruce Krogh, Reid Simmons, and Jeannette Wing. Formal verification of autonomous systems NASA intelligent systems program, September 2001.

[Cri94] Flaviu Cristian. Abstractions for fault-tolerance. In Karen Duncan and Karl Krueger, editors, *Proceedings of the IFIP 13th World Computer Congress. Volume 3 : Linkage and Developing Countries*, pages 278–286, Amsterdam, The Netherlands, 1994. Elsevier Science Publishers.

[Das85] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Trans. Software Eng.*, 11(1):80–86, 1985.

[Dut98] Bruno Dutertre. The Welch-Lynch clock synchronization algorithm, March 1998.

[DY00] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Proceedings of the 12th Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, 2000.

[Ech90] Klaus Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, Berlin, 1990.

[Ets01] Konrad Etschberger. *Controller Area Network. Grundlagen, Protokolle, Bausteine, Anwendungen*. Hanser Fachbuch, May 2001.

[Fle04] FlexRay Consortium. *FlexRay Communications System Bus Guardian Specification Version 2.0*, June 2004.

[Fle05a] FlexRay Consortium. *FlexRay Communication System Preliminary Node-Local Bus Guardian Specification Version 2.0.9*, Dec. 2005.

[Fle05b] Flexray Consortium. *FlexRay Communication System Protocol Specification Version 2.1*, revision a edition, Dec. 2005.

[Fle05c] FlexRay Consortium. *FlexRay Communications System Preliminary Central Bus Guardian Specification Version 2.0.9*, Dec. 2005.

[FMD$^+$00] Thomas Führer, Bernd Müller, Werner Dieterle, Floriam Hartwich, Robert Hugel, and Michael Walther. Time-triggered communication on CAN (Time-triggered CAN – TTCAN). In *Proceedings of 7th ICC'2000, Amsterdam, The Netherlands*, 2000.

[Fou06] Fieldbus Foundation. www.fieldbus.org, 2006.

[Fuc95] Emmerich Fuchs. Cyclic Redundancy Codes (CRCs) for TTP/C. Research Report 10/1995, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1995.

[GB06] D.A. Gwaltney and J.M. Briscoe. Comparison of communication architectures for spacecraft modular avionics systems. Technical Report NASA/TM–2006-214431, NASA, 2006.

[Ham50] R.W. Hamming. Error detection and error correction codes. *The Bell System Technical Journal*, 26(2):147 – 160, 1950.

[HD93] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace Electronic Systems Magazine*, 8:34–39, March 1993.

[HDB05] Brendan Hall and Samar Dajani-Brown. Ringing out fault tolerance. a new ring network for superior low-cost dependability. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 298–307, Washington, DC, USA, 2005. IEEE Computer Society.

[HM$^+$00] Florian Hartwich, Bernd Müller, et al. CAN network with time-triggered communication. In *Proceedings of 7th ICC'2000, Amsterdam, The Netherlands*, 2000.

[ISO94] ISO/IEC. *Information technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*, 7498-1 edition, 1994.

[ISO99] ISO/IEC. *IEEE Standards for Information Technology – Telecommunications and Information Exchange between Systems – Local and Metropolitan Area Network – Specific Requirements – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 1999.

[ISO03a] ISO. *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, iso 11898-1:2003 edition, 2003.

[ISO03b] ISO. *Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, iso 11898-2 edition, 2003.

[ISO04] ISO. *Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication*, iso/prf 11898-4 edition, 2004.

[ISO05] ISO. *Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface*, iso/prf 11898-3 edition, 2005.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[KG93] Hermann Kopetz and Günter Grunsteidl. TTP – a time-triggered protocol for fault-tolerant real-time systems. In *Proceedings 23rd International Symposium on Fault-Tolerant Computing*, pages 524–532, 1993.

[Kon06] Jerzy Konorski. Quality of service games in an IEEE 802.11 ad hoc wireless LAN. In *MSWiM '06: Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*, pages 265–272, New York, NY, USA, 2006. ACM Press.

[Kop91] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. Research Report 8/1991, Technische Universität Wien, Institut fur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1991.

[Kop93] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered? *Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems*, E76-D(11):1325–1332, 1993.

[Kop97] Hermann Kopetz. *Real Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Kop01] Hermann Kopetz. A comparison of TTP/C and FlexRay. Research Report 10/2001, Technische Universität Wien, Institut fur Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[LAK92] J.C. C. Laprie, A. Avizienis, and H. Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[Lis04] Jens Chr. Lisner. A flexible slotting scheme for TDMA-based protocols. In Uwe Brinkschulte, Jürgen Becker, Dietmar Fey, Karl-Erwin Großpietsch, et al., editors, *ARCS Workshops*, volume 41 of *LNI*, pages 54–65. GI, 2004.

[Lis05a] Jens Chr. Lisner. Efficiency of dynamic arbitration in TDMA protocols. In Mario Dal Cin, Mohamed Kaâniche, and Andràs Pataricza, editors, *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2005.

[Lis05b] Jens Christian Lisner. Scheduling in a time-triggered protocol with dynamic arbitration. In *Industrial Electronics, 2005. ISIE 2005. Proceedings of the IEEE International Symposium on*, volume 4, pages 1399 – 1404, June 2005.

[LL84] Jennifer Lundelius and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Symposium on Principles of Distributed Computing*, pages 75–88, 1984.

[Lön99a] Henrik Lönn. A fault tolerant clock synchronization algorithm for systems with low-precision oscillators, 1999.

[Lön99b] Henrik Lönn. Synchronization and communication results in safety critical real-time systems, 1999.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem, 1982.

[MFH$^+$02] Bernd Müller, Thomas Führer, Florian Hartwich, Robert Hugel, and Harald Weiler. Fault tolerant TT-CAN networks. In *Proceedings of 8th ICC'2002, Las Vegas, NV (USA)*, Feb. 2002.

[Mok83] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. Technical report, Cambridge, MA, USA, 1983.

[Moo89] J.F. Moore. ARINC 629, the civil aircraft databus for the 1990s. *Time Critical Communications for Instrumentation and Control, IEE Colloquium on*, pages 5/1–5/2, Oct. 1989.

[Obe05] Roman Obermaisser. CAN emulation in a time-triggered environment, 2005.

[PBG99] Martin Peller, Josef Berwanger, and Robert Griesbach. *byteflight specification (DRAFT) Version 0.5*. BMW AG, Oct. 1999.

[PBG00] Martin Peller, Josef Berwanger, and Robert Griessbach. Byteflight – a new high-performance data bus system for safety-related applications. URL http://www.byteflight.com/presentations/byteflight_paper.pdf, 2000. BMW AG, EE-211 Development Safety Systems Electronics.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, sep 2001.

[RWS04] H. Richter, M. Wille, and C. Siemers. Efficient bandwidth allocation in a ring-based real-time network for automobiles. *mascots*, 00:668–671, 2004.

[Tem98] Christopher Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. *28th International Symposium on FTCS, June 1998, München, Germany*, June 1998.

[Tem99] Christopher Temple. *Enforcing Error Containment in Distributed Real-Time Systems: The Bus Guardian Approach.* PhD thesis, Technische Universität Wien, Dec. 1999.

[TP88] P. M. Thambidurai and Y. K. Park. Interactive consistency with multiple failure modes. In *Proceedings 7th Reliable Distributed Systems Symposium*, Oct. 1988.

[TTA03] TTA-Group. *Time-Triggered Protocol TTP/C High-Level Specification Document Protocol Version 1.1*, 1.4.3 edition, November 2003.

[Upp] Uppaal. www.uppaal.com.

[ZG01] Hongjun Zhang and Pawel Gburzynski. DS-TDMA/CP: A Flexible TDMA Protocol for Wireless Networks. In *Proceedings of the 3rd IEEE International Conference on Mobile and Wireless Communication Networks MWCN 2001*, August 2001.

[Zim80] Hubert Zimmermann. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425 – 432, April 1980.