"Triple-T (Time-Triggered Transmission)" A System of Patterns for Reliable Communication in Hard Real-Time Systems*

Wolfgang Herzner, Wilfried Kubinger, Manfred Gruber
ARC Seibersdorf research GmbH

Abstract. Computer-based hard real-time systems (where "hard real-time" denotes that the missing of a deadline will cause high risk to human lives or environment) become increasingly used in safety-critical application domains like avionics, automotive, or high-speed process control. Crucial to their successful utilisation is a reliable communication among their components as well as strict constraints on transmission times. Triple-T is a group of patterns that not only guarantees transmission durations, but also provides for solving related dependability issues.

Keywords. hard real-time communication, time-triggered communication, dependability, reliability.

1. Introduction

A system is considered to be "real-time", if response (to events etc.) has to occur within given time-spans, making timely response an essential feature [Pont 01] (clause 1.4). Think of a PC, equipped with a framegrabber being able to convert an incoming video-signal into a digitised format like MPEG at a rate of 25 frames per second. If it cannot encode a frame within 40 milliseconds, it will be too slow and eventually loose frames. In contrast, a word processing application on the same PC: of course, the user likes to have it as fast as possible, but besides raising the frustration level of its user above a critical threshold, it will have no effect if the application runs several percent slower than promised or assumed.

As long as a real-time system may miss a deadline without causing significant risk to humans or environment, as can be assumed with the framegrabber example, it is called "soft", otherwise "hard". For instance, electronic brake-controls in cars are considered to be "hard real-time", because untimely reactions may easily lead to accidents.

In general, hard real-time systems become increasingly wide-spread in safety-critical application areas like automotive, avionics, robotics etc. Although hard real-time and dependability are not automatically connected, they are – so to speak – closely related by implication, because only those real-time systems whose failures may cause severe damage are considered 'hard', but those systems are also considered as safety-critical.

Now, the failure of an import hardware component of a safety-critical system may compromise the operability of the whole system. Hence, it must be tolerant against hardware failures. This can (in some cases only) be achieved with redundancy, meaning duplicated hardware components, where a breakdown of some component is compensated by its duplicate. This again nearly enforces communication among the system's components and realisation in a distributed form, at least for allowing negotiation between the duplicates and the rest of the system.

* (c) Copyright 2005 Wolfgang Herzner. Permission is granted to Hillside Europe e.V. to publish and distribute this paper as part of the EuroPLoP conference proceedings

Since we are dealing with hard real-time, it becomes essential that communication among system components also adheres to strict transmission times, at least with respect to hard real-time functionalities. This leads to the topic of this paper: reliable communication with guaranteed transmission times for hard real-time systems.

Such systems are often "embedded" in the sense of being part of larger structures like cars or airplanes. However, the patterns described in this paper do not necessarily presume "embeddedness" of the systems where they are applied.

Another aspect, however, which is less evident, is that many solutions for distributed hard real-time systems try to avoid the concept of a (communication) master. Although redundancy could solve the problem of having such a master as a critical part of the whole system, the implied architectural asymmetry makes it less feasible than symmetrical approaches, where each component (or node) may act as a temporary master in certain situations, if necessary.

After a terminology glossary, we introduce a running example, which will be used in the patterns descriptions to show how the patterns can be applied. Then, the bus architectures for hard real-time communication, where the described patterns have been found, are outlined briefly, followed by the descriptions of the Triple-T patterns.

We consider the presented patterns as a basic set for reliable hard-real-time systems, which will be extended in the future.

Finally, it should be noted that task scheduling for hard real-time applications is not a topic of Triple-T, which concentrates simply on communication.

1.1 Terminology

To ease understanding of the described patterns, common terms are explained in this clause.

Component: A component is an encapsulated building block that is of use when building a large system. Components are characterised by their interfaces with respect to composability and are described by their data properties and their temporal properties [Kopetz 97].

Fail silent: If a system either produces correct results or no results at all, i.e., it is quiet in case it cannot deliver correct service, it is fail-silent [Kopetz 97].

Fault / error / failure: A failure is an event that denotes a deviation between the actual service and the specified service. An error is an unintended incorrect internal state of a computer system. The cause of an error and therefore the indirect cause of a failure, is a fault [Kopetz 97].

Fault hypothesis. Fundamental assumption about the fault-tolerant behaviour of a system. A commonly used is the “single fault hypothesis”: a system must continue to operate correctly as long as not more than one of its components fails.

FCU: A fault containment unit can fail in an arbitrary failure mode without affecting the proper operation of the components not affected by the fault [Kopetz 03].

FTU: A fault-tolerant unit is an abstraction that is introduced for implementing fault tolerance by active replication. An FTU consists of a set of replicated units that produce replica determinate result messages, i.e., the same results at approximately the same points in time [Kopetz 97].

Hard real-time: A real-time system must react to a stimuli from the controlled object within time intervals dictated by its environment. If a catastrophe could result if the deadline is missed, the deadline is called hard [Kopetz 97].

Jitter: A measure for variability of processing or communication actions, e.g. the difference between maximum and minimum durations of a certain kind of action, or the variance of arrival times. It has to be distinguished from delay: a transmission system with high but constant delay has no jitter.

Node: A node is a self-contained computer with its own hardware and software, which performs a set of well-defined functions within the distributed system [Kopetz 97].

*NP-hard / NP-complet*e: The complexity class of decision problems that are intrinsically harder than those that can be solved by a nondeterministic Turing machine in polynomial time. When a decision version of a combinatorial optimization problem is proven to belong to the class of NP-complete problems, which includes well-known problems such as satisfiability, travelling salesman, the bin packing problem, etc., then the optimisation version is NP-hard (e.g. *http://www.nist.gov/dads/HTML/nphard.html*).

Periodic / sporadic / aperiodic/ (task): A periodic task has many iterations, and there is a fixed period between two consecutive releases of the same task. The request time of a sporadic task is not known a priori, and there is a minimum separation between any two requests of a sporadic task. If there is no constraint on the request times of task activations, the task is called aperiodic [Cheng 02], [Kopetz 97]

Task: A task is the execution of a sequential program. It starts with the reading of the input data and the internal state, and terminates with the production of the results and updating the internal state [Kopetz 97].

TDMA: Time Division Multiple Access is a distributed static medium access strategy where the right to transmit a frame is controlled by the progression of real time. To every node which has to transmit data, at least one sending slot is assigned, where it transmits one data frame each. If there are no data to send, an empty frame is transmitted [Kopetz 97].

Validity (time) span: The information content, valid at time of updating the internal state, stays valid for the amount of time defined by the validity span [Poledna++01].

Worst Case Execution Time (WCET). The maximum time some task or processing step may need to execute to completion [Kopetz 97].

1.2 Running Example: Brake-by-wire

As accompanying example for illustrating the patterns, we use the so-called "Electro Mechanical Brake (EMB)". It is considered as the future of a pure brake-by-wire technology for cars since it eliminates brake fluids and hydraulic lines entirely. Furthermore it can include all brake, safety, and stability functions in one system. The principle layout of an EMB is given in Figure 1. On each wheel the braking force is generated by an independent brake node. The speed of the wheel is measured and sent to the pedal node, which is the main control node of the system. This node measures via two sensors the position of the brake pedal, calculates the brake force for each wheel, transmits the new brake force to each wheel node and simulates the real pedal feeling for the driver. A second pedal node exists working in parallel with the first to make sure the system can work if one node goes down.

In this example, we added sensors S8 to S12 to show an additional safety function, namely automatic emergency braking. To realise that, the pedal node needs both the speed of the car and the distance to the next obstacle in driving direction. While S8 and S9 serve to measure the current speed of the car in longitudinal and lateral direction, respectively, S10 to S12 are distance measure sensors for catching the distance to objects in front of the car. If the emergency braking algorithm calculates that a crash will occur definitely (e.g. the distance to an obstacle falls below the “best case” braking distance), it initiates an emergency braking to minimise the harm caused by an accident.

The need for a reliable and time-critical communication in this example (and hence its justification as running example in this paper) should be clarified by following scenario. Assume the electrical connection to one of the brake nodes is interrupted. Since this now isolated node cannot receive the brake force messages anymore, a solution is that it immediately stops braking (as long as it is not in emergency braking state) and the needed brake force is redistributed over the remaining three nodes. This again requires that all remaining nodes, and the pedal nodes in particular, learn about the lost node as fast as possible and with maximum safety.

In contrast to this scenario, the force feedback to the real pedals is not of the same time-criticality (though still has to occur reliably). It will therefore not be considered as important in the rest of the paper.

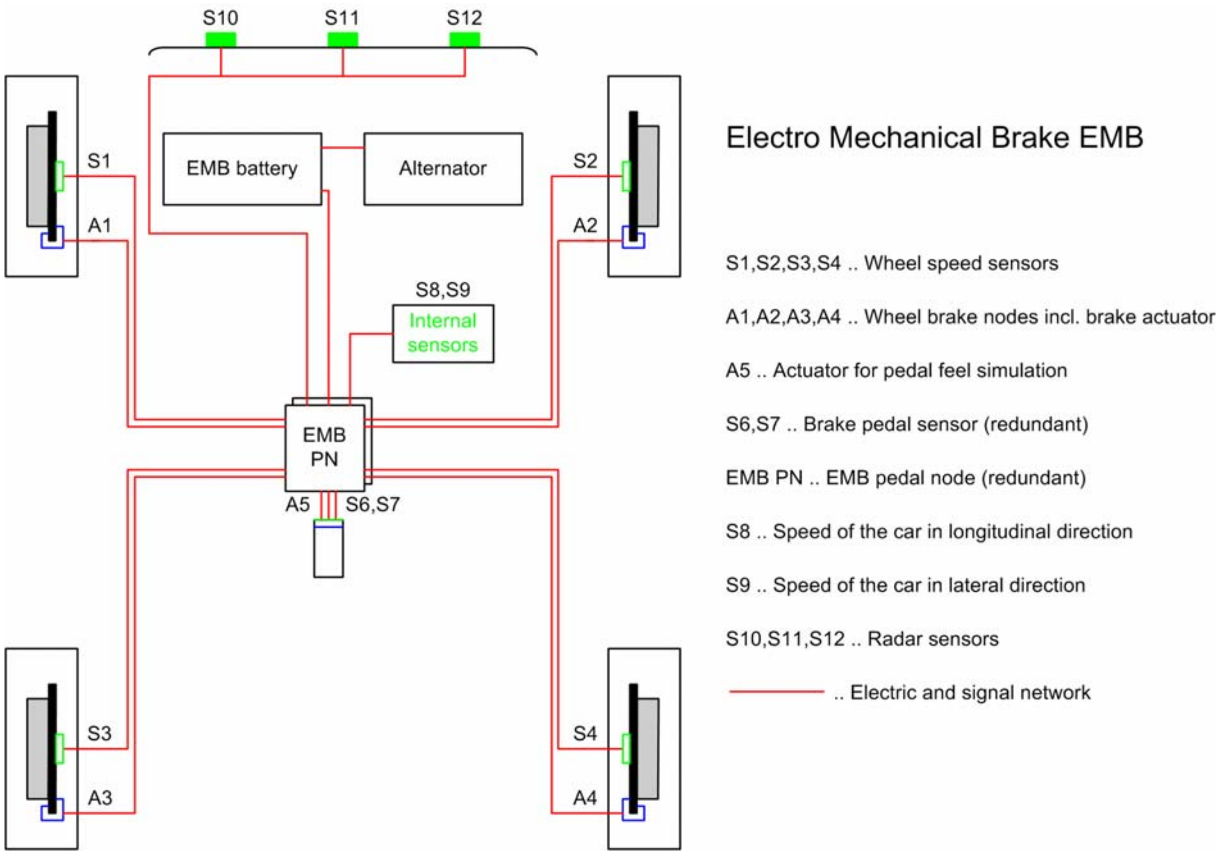
S10
S11
S12
EMB battery
Alternator
S1
A1
S2
A2
S8,S9
Internal sensors
EMB PN
A5
S6,S7
S3
A3
S4
A4
Electro Mechanical Brake EMB
S1,S2,S3,S4 .. Wheel speed sensors
A1,A2,A3,A4 .. Wheel brake nodes incl. brake actuator
A5 .. Actuator for pedal feel simulation
S6,S7 .. Brake pedal sensor (redundant)
EMB PN .. EMB pedal node (redundant)
S8 .. Speed of the car in longitudinal direction
S9 .. Speed of the car in lateral direction
S10,S11,S12 .. Radar sensors
.. Electric and signal network

Figure 1 : Electro Mechanical Brake

1.3 Known Uses

The patterns described in this paper have been found in a number of bus architectures for safety-critical real-time systems. Since they will be repeatedly referenced in the *Known Uses* clauses of the individual patterns, they are shortly outlined in this clause. It can be skipped at first reading, and be revisited whenever wanted.

ARINC 659 (SAFEbus)

The standard ARINC 659 [ARINC 93] was developed by Honeywell under the name 'SAFE-bus' for the Boeing 777, where it serves as Airplane Information Management System (AIMS). It is presumably the most expensive (and mature) time-triggered bus architecture, because not only all bus interfaces are duplicated per node and the bus itself is quad-redundant, the whole AIMS is duplicated.

TTP

TTP (Time-Triggered Protocol) realises the time-triggered architecture (TTA), which has been developed for about twenty years by Kopetz and colleagues at the Technical University of Vienna [Kopetz++94], [Poledna++01]. Although only the interconnect bus is duplicated, it offers a degree of dependability not much lower than that of ARINC 659, due to a number of clever algorithms – where fundamental ones like the clock synchronisation have been formally verified – and the possibility to use node and task replicas in a well supported manner. Applications can be found in the automotive industry, avionics, and special vehicles. At present it supports transmission rates up to 25 Mbit/s.

FlexRay

Developed by a consortium including BMW, Motorola, and Infineon, Byteflight provides up to 10 Mbit/s transmission rates, message priorities which assure deterministic behaviour for high priority messages, and the possibility to mix synchronous and asynchronous transmission [Teepe++04].

A follower of ByteFlight <http://www.ixxat.de/english/produkte/byteflight/byteflight_introduction.shtml>, FlexRay is currently under development by a consortium including BMW, DaimlerChrysler, Motorola, and Philips. It can be regarded as some combination of TTP and ByteFlight: The synchronous data transmission enables time triggered communication to meet the requirement of dependable systems, while asynchronous transmission allows each node to use the full bandwidth for event driven communications. Although full details are not publicly available, it appears that it is intended to provide somewhat less safety and fault-tolerance than TTP to achieve lower costs and higher flexibility. It primarily targets the automotive domain; first prototype hardware components are available.

TTCAN

The TTCAN (time-triggered communication on CAN) protocol is based on CAN. It provides mechanism to schedule CAN messages time-triggered as well as event-triggered. TTCAN is based on the CAN data link layer protocol and does not infringe it at all [Zeltwanger 04]. The main focus of TTCAN is on substituting CAN bus on segments in a car, where time-triggered communication is needed and CAN is already deployed for years (e.g. power-train). TTCAN provides transmission rates up to 1Mbit/s [Führer++00].

Spider

A "scalable processor-independent design for electromagnetic resilience" (SPIDER) has been developed by Miner and colleagues at the NASA Langley Research Centre [Miner 00]. Serving primarily as research platform for recovery strategies for faults caused by radiation induced high-intensity radiated fields and electromagnetic interference (HIRM/EMI), SPIDER supports several bus configurations like bus, star, or central ring. It uses a time-triggered protocol only.

Cost Comparison

Although this paper does not primarily focus on cost issues, it may be of interest to get a rough impression of what has to be paid for a certain level of dependability. Since, however, it is almost impossible to get real cost figures from providers or implementers, only a relative estimation of the deployment costs of the known systems can be is given: CAN (<)< TTCAN < FlexRay < TTP < SAFEbus. Since Spider is primarily a research platform, is has not been considered in that relation. This means, that any of the introduced techniques is (considerably) more expensive than the commonly used CAN-bus [ISO 03], which also shows the lowest level of dependability.

1.4 Patterns-Outline

Triple-T is a system of five patterns, which together establish a base for the development of distributed safety-critical (hard) real-time systems. While fault-tolerance patterns have already been described, e.g. in [Adams++96] and its update [Hanmer++99] for user interfaces in real-time control systems, in [Saridakis 02] for fault-tolerance against system failures or in [Saridakis 03] for fault containment, and patterns for time-triggered embedded systems on dedicated hardware platforms in books like [Pont 01], Triple-T concentrates on reliable communication among nodes in a distributed system with high safety and hard real-time requirements.

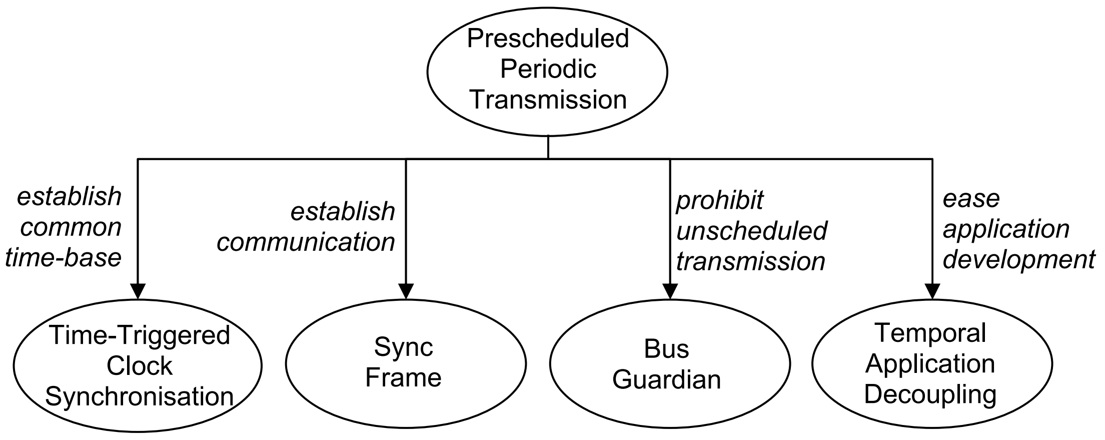
Prescheduled Periodic Transmission
establish common time-base
establish communication
prohibit unscheduled transmission
ease application development
Time-Triggered Clock Synchronisation
Sync Frame
Bus Guardian
Temporal Application Decoupling

Figure 2 : Relationship among Triple-T patterns

PRESCHEDULED PERIODIC TRANSMSISSION is the entry point into Triple-T. It describes how guaranteed transmission times can be achieved to meet hard deadlines. TIME-TRIGGERED CLOCK SYNCHRONISATION provides a solution for keeping the local clocks synchronised without central master. SYNC FRAME addresses how the whole communication can be initiated, as well as how nodes can enter an already running communication. The BUS GUARDIAN is necessary for keeping faulty nodes from disturbing the communication among healthy nodes. Finally, TEMPORAL APPLICATION DECOUPLING describes how application(task)s can be decoupled from the communication process to ease both their development and their scheduling.

In the rest of the paper, these patterns are described.

2. Prescheduled Periodic Transmission

Also known as: TIME-TRIGGERED COMMUNICATION, TIME-DIVISION MULTIPLE ACCESS (TDMA)

2.1 Context

Hard real-time systems which have to be distributed for fault-tolerance reasons, and where the communication among the components shows the following characteristics:

- Most if not all of the messages to be exchanged between components are small (e.g. single sensor measures), but have to be frequently transmitted.
- Transmission times must be guaranteed (a consequence of being hard real-time).
- Break-down of a node must be detected by the remaining nodes within a given time (a consequence of being fault-tolerant).

2.2 Example

Consider the brake-by-wire system in a car as discussed in the introduction. Here, typical components are the sensors for the brake-pedal position, wheel angle speeds, vehicle speed (longitudinal and lateral), and distance measures to frontal obstacles. The computing components analyse the situation in front of the car from speed and distance measures, they evaluate the resulting brake forces on all wheels, and the actuators turn the latter force information into physical brake forces, as well as actuators provide feed-back to the driver by adjusting both position and resistance strength of the brake-pedal. Except for the driver feedback, all corresponding information has to be transmitted frequently with assured transmission duration, usually every few milliseconds, but are of small and fixed size. And whenever a node ceases to function, this has to be recognised by the remaining nodes in the same time range, to adapt the system's behaviour immediately to the new resource situation (e.g. by redistributing the needed brake force over the remaining three brake nodes).

2.3 Problem

How can required transmission times be guaranteed, when many, mostly small, messages have to be sent repeatedly, and it has to be recognised as soon as possible, whenever some node fails to deliver its service?

When using conventional event-driven communication, where each node sends a message as soon as it has one to transmit, we are faced with several problems. A first problem is that it is almost impossible to guarantee requested transmission times, due to the risk of collisions on the bus. If, for instance, two nodes start to send almost simultaneously, so that they have not yet recognised each other, both messages are corrupted and have to be retransmitted, even raising the danger of another collision. A well-known consequence of this effect is that (non-switched) Ethernet loads of more than about 30% of its maximum capacity significantly raises transmission times.

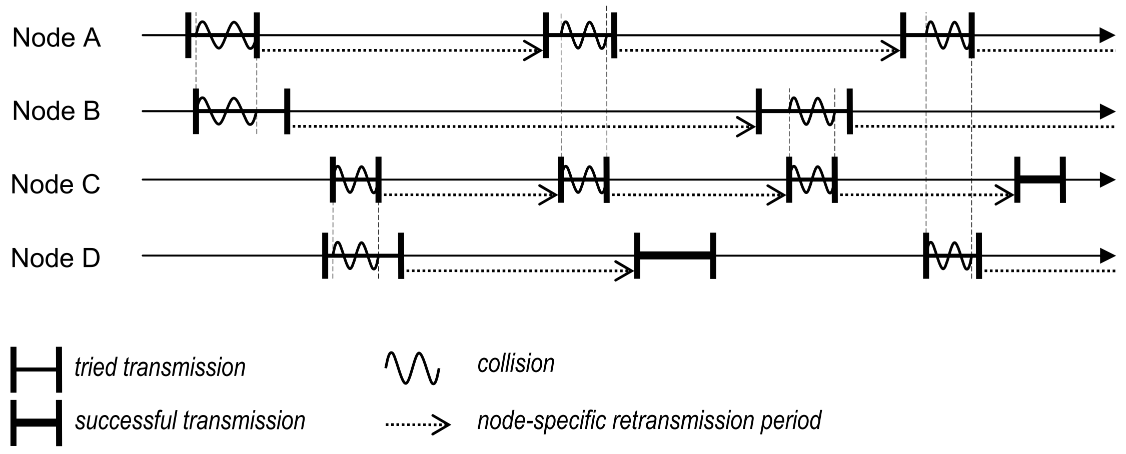

Figure 3 : Example for collisions on Ethernet

Figure 3 illustrates this effect. Nodes A and B send almost simultaneously, so their transmissions collide and are disturbed. Then, each of them waits a node-specific time before it retransmits. In the meantime, the same happens to nodes C and D. Due to their individual retransmission times, C collides with A at its next try, while D succeeds. Happy about its successful transmission, it starts to transmit a new message, which collides with the third try of A, while C succeeds at its fourth try. So, in the shown interval, there have been 12 transmission tries with only 2 of them successful.

Another concern is that nodes may go out of function in various ways. If they are able to send a corresponding information to the other nodes before shutting down, the remaining system can adjust to the new situation. If, however, a nodes ceases without warning, the other nodes may recognise this only by detecting that they are not receiving messages from that node anymore. (This presumes that any node sends at least one message regularly, even pure actuator nodes.) Although this is possible, it has to take potential transmission delays due to collisions as discussed before into account, which may cause unacceptable delays or jitter.

Finally, a rather minor issue is that we need headers for each message – containing the sender identification, the type/meaning of the transmitted information, sending/creation time, and possibly the intended recipient(s) if we do not broadcast – which tend to need significantly more bits than the actual data. If, for instance, only one data byte is needed, e.g. representing the brake pedal position in percent of "down", a major part (two thirds or more) of the available bandwidth is consumed by framing information, but not by relevant data. If we take the first problem into consideration as well, then we have to conclude that in the given context the net utilisation of the available bandwidth must stay clearly below 10% to be able to assure a requested transmission time with a probability of about 99% or more – which is still below the 10^{-9} failures/hour required for hard real-time systems.

Why not using Token Ring?

An alternative to Ethernet-like transmission could be a token ring (e.g. *http://www2.rad.com/networks/1996/toknring/toknring.htm*). Here, all nodes are connected in a ring-like network, where a so-called *token* is passed from one node to the next. The token is some sort of message, where other messages may be appended. When a nodes receives the token, it removes all messages dedicated for itself, and appends messages to be sent to other nodes, and forwards the token to the next node. Since only that node holding the token is allowed to send, collisions are avoided.

As long as nodes behave correctly, bandwidth utilisation can be significantly higher than with Ethernet. However, a single faulty or slow node can cause at least significant delays. In addition, nodes which erroneously start to transmit without token may cause collisions and severely hamper the communication. And even without such problems, transmission times are difficult to guarantee, because transmission times between two specific nodes depend on the number of nodes between them.

As a consequence, token ring does not represent a reasonable solution to the given problem.

2.4 Solution

Use a periodic transmission schedule, where each message has at least one time-slot assigned where it is allowed to be transmitted, and with a period which is short enough to fulfil the shortest needed transmission time within the system. Furthermore, define for each message the precise phase within the period when it has to be sent.

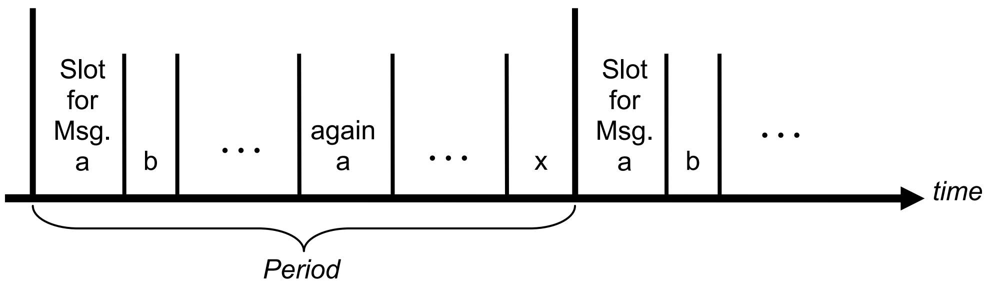

Figure 4 : Example for periodic transmission schedule

As soon as time reaches the slot of a certain message, the respective sender node (or the corresponding tasks, respectively) sends the data exactly as prescribed by the schedule, independent of whether it changed since the previous period or not. Since the meaning of each data sent is completely described by its phase within the period, it is not necessary to send a header with it; consequently, only the pure information need to be contained in a message. Data which have to be transmitted more frequently than others may have more than one time slot assigned within a period, as indicated with message a.

(Use BUS GUARDIAN for prohibiting nodes from transmitting outside of their slots.)

Basically, this pattern implies broadcast communication rather than multicast or even point-to-point communication. Of course, each node listening on the bus can ignore data not relevant for it or the tasks running on it, respectively.

This communication is called "*time-triggered*" (opposite to *"event-triggered*"), because the progression of time is the only reason for a transmission.

Sometimes, the set of all transmitted information within a period is considered as "public state" of the system, which gets continuously updated, such that after each round each node within the system conceptually has knowledge of the most recent state values.

2.5 Implementation

The first step is the specification of the schedule, which has to occur during the design of the overall system. Following questions have to be considered when preparing the schedule.

Constraints for the Schedule

Highest transmission frequency needed.

This will define the shortest period. It essentially depends on the highest possible and desired refresh rate for any information to be transmitted.

Lowest refresh rate needed.

Some data may be generated with frequencies significantly lower than the shortest transmission time needed. If, for instance, some measure value cannot be read more than four times a second, but needs a greater amount of bytes to be transmitted (which, for example, could apply to visual sensor data) it can make sense to consider this in the schedule.

Validity time spans.

Messages have to be scheduled such that the sum of the time span the corresponding data stay in the sending node and in the receiving node(s) until being processed, as well as the transmission time itself do not exceed the respective validity time span.

Aperiodic messages.

It may be necessary to allow transmissions of aperiodic or even sporadic messages, for example diagnostic information.

Future extensions.

If the available bandwidth is not already exhausted, it is wise to anticipate the need for new messages within existing slots as well as for new slots for future additions, by adding free ‘space’ to both time slots and transmission period. The reason is that as long as future extensions do fit into the free time slots, no re-design is needed for the existing communication, because it is safely not affected by these extensions.

Available bandwidth.

Of course, the available bandwidth affects the free space left over for aperiodic transmissions and future extensions.

Preparing the Schedule

The highest needed transmission frequency determines the shortest period within the schedule. To leave room for positioning messages within the schedule with respect to their validity time spans, it is sensible to take even a shorter period than implied by the highest needed transmission frequency.

If there are transmission frequencies significantly lower than the highest one, as indicated by the lowest refresh rate needed, it may make sense to use more than one period, resulting in a hierarchical period system, where the number of shorter periods is an integer multiple of the longer one. For instance, the shortest period is 10ms, while the longest is 250ms, resulting in 25 short periods within one long. Of course, that hierarchy may consist of more than two levels.

Message slots are now placed within the periods in a way that validity time spans are not violated. In particular, so-called "*processing pipes*" have to be considered: if task A (e.g. a sensor task) on node 1 sends message X to task B (a computing task) on node 2, which processes it and sends message Y to task C (an actuator task) on node 3, then X and Y have to be scheduled such that both B can process X before its validity time span expires and produce Y right in time so that it can be processed by C before Y's validity time span is exceeded.

If time slots for aperiodic messages are needed, they are scheduled in unused spaces of the schedule prepared so far. Since these slots shall serve for transmission of data from various nodes, a special protocol is needed to control communication within such time slots.

Finally, free transmission time is computed by subtracting all time slots needed so far from the available transmission bandwidth.

See "Running Example Resolved" for a diagram showing a possible schedule.

Using the Schedule

The schedule will finally be installed at each node of the system in a way that the transmission control unit of each node knows the whole system schedule at runtime, or at least those parts relevant for it.

2.6 Running Example Resolved

We start with listing the assumed frequencies with which individual data are or can be provided.

Datum	**Unit**	**Source**	**Size (Bytes)**	**Min. Time (ms)**[†]
Brake pedal position	mm.	S6, S7	1 (unsigned int.)	44
Wheel speed (front left … rear right)	°/sec	S1 … S4	2 (signed int.)	2

[†] Minimal sampling time for sensors; WCET for control algorithms

Car speed longitudinal	cm/sec	S8	2 (signed int.)	5
Car speed lateral	mm/sec	S9	2 (signed int.)	18
Minimum front distance	Cm	S10 … S12	2 (unsigned int.)	95
Front distance credibility	0→no, 1→ok	S10 … S12	1 bit	95
Wheel brake force (per wheel)	milliNewton	PN1, PN2	2 (unsigned int.)	9
Brake pedal feed back position	Mm	PN1, PN2	1 (unsigned int.)	9
Brake pedal feed back force	milliNewton	PN1, PN2	1 unsigned int.)	9

Table 1 : EMB-example, characteristics of transmitted data

If "front distance credibility" is 1, the "minimum front distance" shall indicate the smallest measured distance to an obstacle, otherwise, the latter value is of no use. (Although this 'causality' suggests to transmit "minimum front distance" as event rather than time-triggered, we keep it as time-triggered for simplicity.) The 9 milliseconds for data provided by the EMB processing nodes are assumed to result from strong WCET (worst case execution time) estimations. Actuators are not considered, because they only receive data, but let assume that the wheel brake nodes cannot process brake force commands more often then every 10 ms.

Obviously, the maximum of the PN's 9 ms and the actuator nodes' 10 ms define a basic transmission frequency; however, it is not the best way to force the whole data exchange into one 10 ms period, simply because some data like brake pedal position cannot be provided so often. But the longer intervals can appropriately be fit into multiples of this period, making a hierarchical schedule as shown in Figure 5 plausible. There, messages are denoted by their source sensor, or "a"…"c" for the processor node's output as given in the order of Table 1.

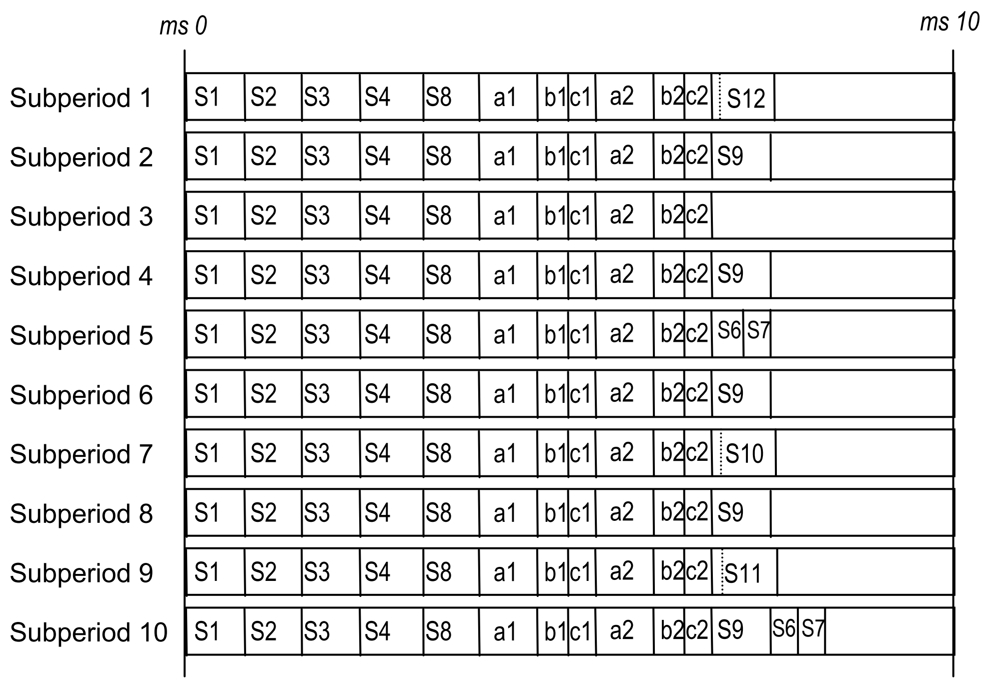
ms 0
ms 10
Subperiod 1 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S12
Subperiod 2 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S9
Subperiod 3 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2
Subperiod 4 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S9
Subperiod 5 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S6 S7
Subperiod 6 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S9
Subperiod 7 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S10
Subperiod 8 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S9
Subperiod 9 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S11
Subperiod 10 S1 S2 S3 S4 S8 a1 b1 c1 a2 b2 c2 S9 S6 S7

Figure 5 : EMB example schedule

The period consists of ten subperiods of 10 ms each, resulting in a period of 100 ms. For each message, an own slot is defined. Every message with a delivery period up to 10 ms is transmitted in each subperiod, and the lateral car speed every second subperiod. The other, less frequently provided messages are distributed such that the lengths of the subperiods is nicely balanced, resulting in a net bandwidth of less than 20 bytes per subperiod, or some 20 Kbit. Even with modest bus capacities, there is enough spare space per subperiod for future extensions, indicated by the blank areas at the end of each subperiod.

2.7 Consequences

Benefits

Guaranteed transmission time. Transmission time of a message within a time-triggered slot can be guaranteed to the frequency it is scheduled, because due to the schedule each node sends only when it is planned and therefore no transmission collisions occur. (See BUS GUARDIAN for assuring that this is actually the case.)

Fast detection of erroneous nodes. That a node misses its slot is conceptually immediately detected by the other nodes. Hence, fail-silent nodes can inform the remaining nodes in the system simply by not sending anymore. This is a significant advantage of time-triggered communication over event-triggered one, which needs various additional mechanisms for detecting erroneous nodes like *I AM ALIVE* or *YOU ARE ALIVE* [Saridakis 02].

Minor message header needed. Since message content is completely defined by its position within the schedule, no information about sender, content type, creation time and alike need to be attached to the message. Though, some framing information may be needed to let both receiving communication controllers detect transmission at all, and to safely distinguish data messages (data frames) from synchronisation frames (see TIME-TRIGGERED CLOCK SYNCHRONISATION). However, for small messages like sensor values or control values for actuators, which are often represented by single or small sets of bytes, this may be a significant reduction of transmission overhead.

High bandwidth utilisation. Since within the time-triggered parts of the schedule no collisions occur, bandwidth can be utilised up to almost 100%.

Composability. If new messages which need to be added can be placed into free slots (unused spaces), then the communication can be extended without any risk of affecting existing transmissions, which cannot be guaranteed with event-triggered transmission. With only the latter available, communication has to be re-designed (re-tested, re-certified) as a whole every time messages are added.
Of course, this argument applies only to transmission, while all affected tasks (that which sends the new message and all which read it) have to be re-tested/certified. But if a new subsystem is added, which uses its own messages, then only this subsystem as a whole has to be tested and certified.

Liabilities

Synchronisation needed. Time-triggered communication relies on synchronisation of all participating nodes, which has to be kept much stronger than in event-driven communication. TIME-TRIGGERED CLOCK synchronisation provides a solution for that, based on

PRESCHEDULED PERIODIC TRANSMSISSION, while SYNC FRAME helps to establish communication at all.

Risk of transmission disturbance. Fault nodes can easily disturb communication by transmitting out of phase. BUS GUARDIAN helps to avoid this.

More complex application development. Application tasks must adhere to the transmission schedule with respect to sending and receiving data. TEMPORAL APPLICATION DECOUPLING tells how this complexity can be significantly reduced.

Inflexibility. The time-triggered parts are highly inflexible at run-time. For instance, communications which need more than half of the bandwidth to be occupied by one sender at some time, and to be occupied by another node at some other time, cannot be realised with such transmission schedules defined at design time. Of course, it is possible to switch between different schedules, but this is hard to implement with respect to maintain dependability during schedule switching.

High design effort. Developing correct schedules is an NP-hard problem. Currently no methods exist which can evaluate the optimal schedule for any given set of constraints. Instead, so-called heuristic schedulers are used which may fail to find a solution even if one exists.

Transmission overhead through unnecessary transmission. It can be argued that the avoidance of message headers is traded against transmission more often than necessary. This will be the case if constraints laid upon the system designer by the used developing tools do not allow e.g. various transmission periods or the use of aperiodic (event-triggered) messages in dedicated time slots.

Power consumption. Highly periodic transmission may consume more power than event-driven transmission. In power-aware environments, this could become a critical issue.

2.8 Known Uses

All bus architectures described in the introduction support time-triggered transmission. Besides TTP, where the periodic transmission schedule is named MEDL (Message Description List), and Spider, all also support event-triggered transmission, in principle in the described way. TTP provides two levels of schedule hierarchy: a so-called "bus cycle" can consist of a power of two "TDMA rounds", where all rounds are of equal length and layout with respect to node-specific time slots, but within each round, each node may send different messages (or none at all) within its time slot.

FlexRay differs slightly from the others in the way that each node gets only those parts of the schedule loaded, which concern messages it is interested in. This has the advantage of a somewhat higher flexibility, because on adding a new node to an existing system with exploiting composability, the schedule in existing nodes need not be updated, but at the cost of a higher start-up complexity; see Known Uses clause of SYNC FRAME. Likewise, a TTCAN controller only gets the necessary information it needs for time-triggered sending and receiving of messages as well as for sending of spontaneous messages.

2.9 Related Patterns

For executing both application tasks and system tasks serving for transmission, the *CYCLIC EXECUTIVE* pattern or – in special cases – the *ROUND ROBIN* pattern can be applied [Douglass 03]. *CO-OPERATIVE SCHEDULER* [Pont 01] can be used on the operating system level to execute tasks in alignment with the transmission schedule.

3. Time-Triggered Clock Synchronisation

3.1 Context

Hard real-time systems which have to be distributed for fault-tolerance reasons, where PRE-SCHEDULED PERIODIC TRANSMISSION is applied to guarantee that requested transmission times are not violated, and where a central clock is not available.

3.2 Problem

Each node has a local clock oscillator. Even optimally calibrated clocks on different nodes will diverge due to different physical and environmental parameters. But keeping the clocks of all nodes synchronised is crucial for a successful time-triggered communication. How can this be achieved in the absence of a central clock, with sufficient precision for the requested transmission rates?

3.3 Example

Consider our brake-by-wire application. Here, we have got five distributed nodes, each of them using an own oscillator. To keep hardware costs low it is common practice in cost-sensitive applications to deploy cheap quartz oscillators. These devices feature a low frequency stability and a high temperature drift. Therefore some kind of clock synchronisation is necessary.

3.4 Solution

Compare precisely the actual arrival times of messages received from other nodes with the scheduled arrival times, and use an error compensating function of these differences to correct the own clock.

The solution is based on the transmission schedule. As soon as a node has successfully joined the communication on a time-triggered bus, it can simply compare the arrival times of the messages within time slots of the other nodes with the corresponding times given in the schedule. If, for instance, on a node *K*, message *a* from node *1* arrives two ticks later than scheduled, message *b* from node *2* five ticks later, and message *c* from node *3* one tick earlier, then *K* can deduce that it is a bit too early (i.e. $(2+5-1)/2 = 3$ ticks) and has consequently to delay its own clock for that value.

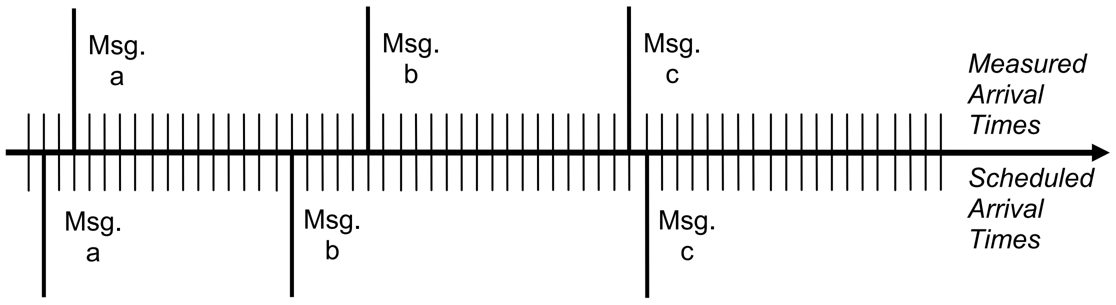

Figure 6 : Comparison of scheduled with measured message arrival times

If this local clock correction is applied in each node, all clocks are kept aligned up to a certain maximum of tick differences. For instance, node 3 would probably delay its clock a little bit as well, while nodes 1 and 2 would speed up their clocks.

3.5 Implementation

Since TIME-TRIGGERED CLOCK SYNCHRONISATION is vulnerable to "wild clock readings"[‡], it is desirable to exclude these from the clock correction calculation. This could be done by limiting the allowed difference between scheduled and measured arrival time to a maximum value. Nodes whose messages arrive too early or too late (i.e. the corresponding arrival time differences exceed this limit), not only are not considered for clock correction, but this can also be a hint that these nodes are faulty. This limit can even be used to judge the own clock: if all (or most) readings from the other nodes are uniformly too early or too late, this indicates that the own clock has gone wild rather than all the others.

A variant is the Welch-Lynch algorithm, also known as "t-fault-tolerant midpoint" [Welch++88]. If up to t faulty nodes shall be tolerated by $n+1$ nodes in the system, then the differences are increasingly or decreasingly ordered, and the average of the $(t+1)^{st}$ and $(n-t)^{th}$ value computed. Of course, this requires $n > 3t$.

Once a necessary clock correction has been identified, several methods exist to apply it. If the own clock counter (i.e. the mapping of quartz periods to clock ticks) can be adjusted, then it is feasible to apply the correction by appropriately adjusting this mapping; this has the advantage of a maximally smooth clock correction. If, however, such an adjustment is not supported, excess ticks must be skipped and missing ticks must be added. This could be done at begin of the next transmission period. However, since such a correction is a 'hard' or 'jumping' clock correction on the one side, and has to affect the whole node including operating system and application tasks, jumps of several ticks may be dangerous. It appears therefore preferable to distribute the insertion or omission of clock ticks over a longer period of time.

This leads to the question of the frequency with which these measurements and corrections shall be executed. The highest frequency would be that of the transmission schedule; that means that each transmission round the arrival time deviations are measured and the resulting

[‡] This term denotes effects caused by an erroneous node clock, which runs significantly faster or slower than those of other nodes in a system, or even irregularly.

clock correction computed. Since the necessary calculation are rather simple, and the resulting corrections can be expected to be very small, this appears to be the best solution.

3.6 Consequences

Benefits

No further mechanism needed. Since this pattern relies on the periodic transmission schedule, it can be implemented without the need for further communication like distributing specific clock-sync messages.

Fault-tolerance included. Too large arrival time deviations indicate faulty nodes automatically. Likewise, methods like Welch-Lynch automatically avoid poor clock sync due to wild clock readings.

Liabilities

No sync with external time. All clocks will synchronise to the average of all participating clocks. If this average deviates significantly from standard time, the whole system will drift apart from standard time.

Minimum number of nodes needed. If the clock synchronisation shall be tolerant against *t* simultaneously faulty nodes, then at least *2t+2* nodes are needed (*2t* for ignoring *t* extreme measures on both ends, *1* for the good reading, and *1* recipient).

3.7 Known Uses

TTP uses Welch-Lynch. However, there is a distinction between nodes with ordinary clocks and such with accurate clocks (expensive), and only the latter are considered.

FlexRay also uses the standard Welch-Lynch.

3.8 Related Patterns

An alternative is to use a "central clock" as described in the *SHARED-CLOCK (S-C) SCHEDULER* [Pont 01] (p.543-5), where a master sends tick messages at regular intervals within the framework of the transmission schedule, and all slave nodes adapt their time to these master ticks. As already discussed, this is vulnerable to faulty masters, as long as no smart fault-tolerance mechanisms like duplicating fail-safe masters are applied. However, a benefit of this pattern is it can be coupled with an external clock, thus allowing to align a whole cluster with standard time.

4. Sync Frame

4.1 Context

Distributed hard real-time systems, which have to be distributed for fault-tolerance reasons, and which use PRESCHEDULED PERIODIC TRANSMISSION to guarantee that requested transmission times are not violated.

4.2 Problem

Once a node is started (or restarted after a failure) and ready to join the communication: how does it synchronise with the other nodes, i.e. how does it recognise the current transmission phase? Since transmitted information contains essentially application specific data and presumably error correcting code like CRC for transmission error detection, it is difficult for a node which wants to join to interpret the received bit sequences correctly.

A similar problem exists at start-up of the whole system, when initially no transmission occurs at all. How one gets the periodic transmission finally running? A special constraint is that start-up should work without master, as argued in the introduction.

4.3 Example

Consider our brake-by-wire application. When the ignition key is turned, all nodes boot and after initialisation and self-test they are ready to communicate. However, since no node is sending so far, how can the synchronised communication become established, which is necessary for running the pre-defined transmission schedule? Note that without an established communication, the local node clocks are not yet synchronised as well.

4.4 Solution

Let special messages be transmitted, which unambiguously identify the phase within the transmission schedule when they are sent. These messages are called “synchronisation frames”, or "sync-frame" for short.

In detail, the solution consists of several steps.

First, define a sync-frame, containing a unique bit pattern, which can safely be distinguished from application data, and additional information like the actual phase within the schedule when it is sent.

Second, for each node – or at least a number of nodes, see Implementation clause, reserve a special part of its time slot where it will transmit the sync-frame, usually at begin of the slot. If a hierarchical schedule is used, then this may be needed only on the highest level.

Now, an "out-of-sync" node which wants to join a running communication, listens until it recognises the bit pattern of the sync-frame. From the additional information it learns the current phase within a transmission period, which allows it to settle its own transmission control.

If it does not receive a sync-frame for a certain period of time, e.g. a bit longer than the longest period within the predefined schedule, then it starts sending such a frame by itself. If other nodes, also being in the start-up phase, receive this sync-frame, they can synchronise and start to transmit. This again is detected by the sender of the sync-frame, which can now start to transmit in its time slots, and the communication is established. If no node is responding, either because no other node is ready, or due to a collision on the bus, the sending of sync-frames is usually repeated.

If two starting nodes send sync-frames simultaneously, these will be corrupted and not successfully received by any node of the system. To avoid that the same collision appears again, each node must wait a different time, such that, in the worst case, after any possible combination of collisions among nodes, finally one will succeed. See next clause, Implementation, for how these different waiting times are determined.

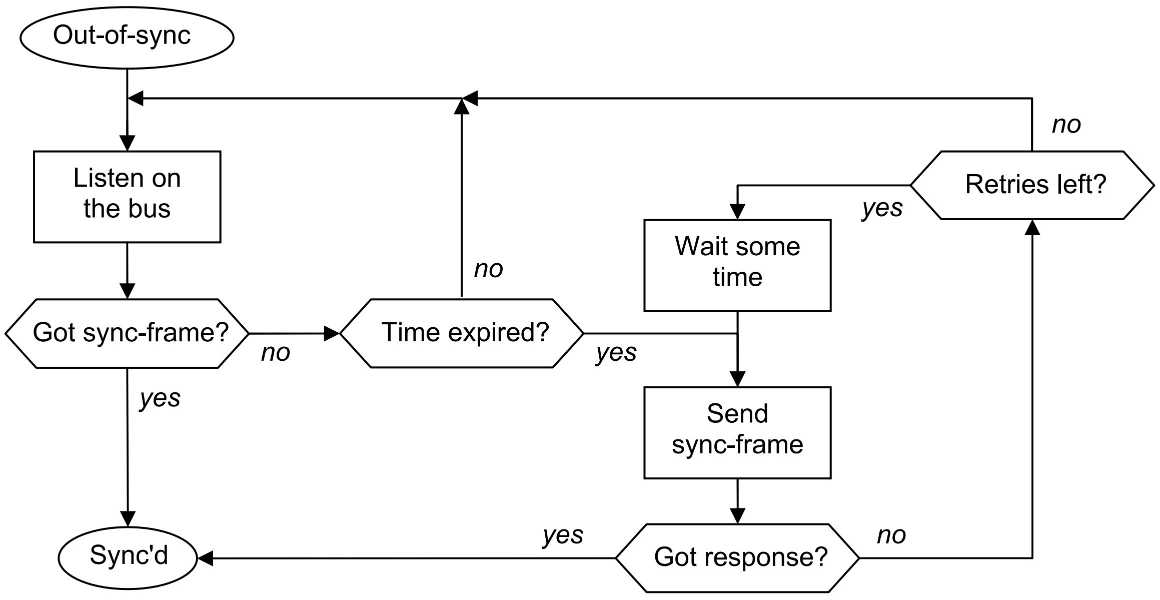

Figure 7 : Control diagram of SYNC FRAME

In Figure 7, “Sync’d” means that the node knows the current phase within the transmission schedule, and can therefore start to send and/or receive messages. “Send sync-frame” includes initiating local processing of the transmission schedule; hence, a sync-frame is sent at precisely that phase it is scheduled for the given node. Consequently, “Got response?” = “yes” means that the sync-frame sending node receives valid messages from other nodes. This indicates to it that other nodes have successfully received its sync-frame and established their transmission schedules in synchronisation with its own.

4.5 Implementation

How the identification bit pattern of sync-frame looks like, depends on the general encoding of application data and other information on the bus. Since some form of redundancy will be needed to detect transmission errors, e.g. CRC or some other error correcting encoding, application data are usually embedded into – though small – control frames, or encoded. This allows for reserving specific patterns for the sync-frame. In most cases, quite a few bytes are needed for it.

It is not necessary that each node sends sync-frames during the normal transmission period. Rather, it has only to be guaranteed that even in the assumed worst case at least one healthy node remains which sends sync frames. If *m* nodes are accepted to fail simultaneously, then obviously at least *m+1* nodes must transmit sync frames. With the common "single fault hypothesis", this means at least two nodes. Therefore, Figure 7 shows how nodes which have sync-frames scheduled behave at startup (we will call those nodes *sync-nodes*); those which have no sync-frames scheduled simply listen.

An important pre-requisite for efficient application of SYNC FRAME is that the waiting periods of sync-nodes between two consecutive sync-frame transmissions are sufficiently different for all nodes. In particular, no waiting period must be an integral multiple of some other within a system. Since these periods are defined prior to runtime (e.g. at installation or even at construction time), it is straightforward to fulfil this request. For instance, consecutive odd numbers where the product of the smallest with the number of sync nodes is larger than the largest could be used. Then, a sync-node will safely transmit a sync-frame successfully without collision latest after one collision with each of the other sync-nodes in the system. Of course, the number of retries (see Figure 7) has to be larger than the number of sync-nodes. As a consequence, the individual waiting should be determined during scheduling.

This constraint has to be considered whenever a system is reconfigured; for example, when during maintenance in a car an old node is replaced by a new one, or even new nodes added.

A risk of this pattern is that a faulty node may be incapable to detect sync-nodes or even any communication although it is already going on. In this case, it will start to send sync-frames without need. To avoid this, further mechanisms are necessary, like BUS GUARDIAN.

A further risk are so-called “Byzantine” situations at start-up. This includes all situations where either a node switches irregularly between faulty and non-faulty behaviour, or the healthiness of a node is judged differently by different nodes. Many possible faults can be detected by healthy nodes (by sophisticated extensions of the algorithm sketched in Figure 7 or by methods not addressed by Triple-T) and corrected by the whole system as long as the underlying fault hypothesis is not violated. However, if not all components are fail-silent, it is impossible to safely exclude undetectable and hence unsolvable Byzantine situations. For a more thorough discussion of this topic, have a look at corresponding literature, e.g. [Driscoll++03].

A possible effect could be that several sync-frames emitting nodes try establishes successfully communication among a subset of the whole cluster, thus creating so-called “cliques”

4.6 Running Example Resolved

In our running example, we assume that it is sufficient to transmit sync frames once per period, and by two nodes. We choose the processor nodes (PN1, PN2) for that. Since currently in sub-period 3 the fewest messages are scheduled (see Figure 5), we add the sync frames there at the begin of the time slots of PN1 and PN2, namely before a1 and a2, respectively, as indicated by the grey boxes in Figure 8.

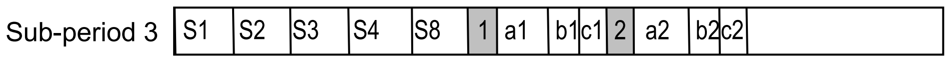

Figure 8 : Subperiod 3 with sync frames

For the content of these sync frames, we choose a unique bit pattern, followed by one bit indicating the number of the processing node it is sending, i.e. '0' for PN1, and '1' for PN2. And we choose 111 and 113 ms as their waiting periods between sync-frame transmissions.

4.7 Consequences

Benefits

Decentralised start-up. There is no need for a master to initialise communication.

Speed. A new (or restarted) node may join within two periods. And as long as no series of increasingly unlikely sync-frame collisions occur, start-up of a whole system will occur within a few periods.

Self-stabilisation. As long as no faulty or malicious node disturbs working transmissions of good nodes, this pattern finally integrates all good nodes into a communicating system.

Liabilities

Limited fault-tolerance. In presence of faulty nodes, communication may not settle or being significantly hampered, as long as no further mechanisms are used to suppress erroneous sync-frame sending of such nodes. BUS GUARDIANS allow for coping with deterministically detectable faulty nodes.

Increased complexity. Sync-frames have to be considered during evaluation of the transmission schedule, as well as different waiting times for all nodes be computed.

Reduced bandwidth. Sync-frames don't carry application-relevant information and therefore reduce the bandwidth available for that. However, since sync-frames are rather short and need to be sent too often, the bandwidth loss is usually about 1% or less.

4.8 Known Uses

In ARINC 659, actually two flavours of sync-frames are used. "Long Resync" frames are transmitted during a running communication. Hence, an out-of-sync node first listens for such a "Long Sync"; if it fails to recognise any for a certain length of time, it send an "Initial Resync", indicating to the other nodes that it wants to restart the whole communication.

On TTP, sync-frames are called "I-frames". If they are transmitted in a time slot together with application data, the whole frame is called "x-frame". In addition, clever algorithms are implemented to minimise the danger of Byzantine errors during start-up.

Since FlexRay-nodes get only their own part of the transmission schedule loaded, they have to learn the full configuration by observing message traffic during start-up, which again makes communication initialisation more complicated and vulnerable to misbehaving nodes.

This risk is reduced by restricting the creation of sync-frames to dedicated nodes, which then can be built more expensive, but also more reliable than the majority of the remaining nodes. Of course, this approach weakens the symmetry property intrinsic to SYNC FRAME.

4.9 Related Patterns

PRESCHEDULED PERIODIC TRANSMISSION is a precondition for applying SYNC FRAME. BUS GUARDIAN helps to avoid sending invalid sync-frames by faulty nodes during rejoin.

5. Bus-Guardian

5.1 Context

Hard real-time systems which have to be distributed for fault-tolerance reasons, and where PRESCHEDULED PERIODIC TRANSMISSION is applied to guarantee that requested transmission times are not violated.

5.2 Problem

How can it be guaranteed that a node or its communication handler, respectively, actually sends data on the bus only in the time slots reserved for it? In particular, how can it be assured that an erroneous node does not disturb the communication among correct nodes by "babbling" into their time slots? (In general, a "babbling idiot" failure occurs, if an erroneous node does not adhere to the rules of the communication protocol and sends messages at arbitrary points in time.)

5.3 Examples

Consider, for instance, a node basically following the transmission schedule of the whole system it is member of. However, due to an electrical fault (perhaps induced by a transient electro-magnetic field disturbance), it sporadically produces noise on the bus outside its own time slots.

Another typical problem are so-called "slightly out of specification" – or "SOS" – errors, caused when digital devices work close to the voltage border between the representation of '0' and '1'. If, for example when using TTL levels, '0' is represented by 0 Volt and '1' by 5 Volt, and the border is 2 Volt. Then a device which looses power will increasingly represent '1' close to this border. This may result not only in transmission errors – which should be detected by recipients, but also in complete misbehaviour of that device, which again can turn it into a "babbling idiot".

5.4 Solution

For each node, provide a separate device – the "bus guardian" – which restricts transmission to the nodes own time slots.

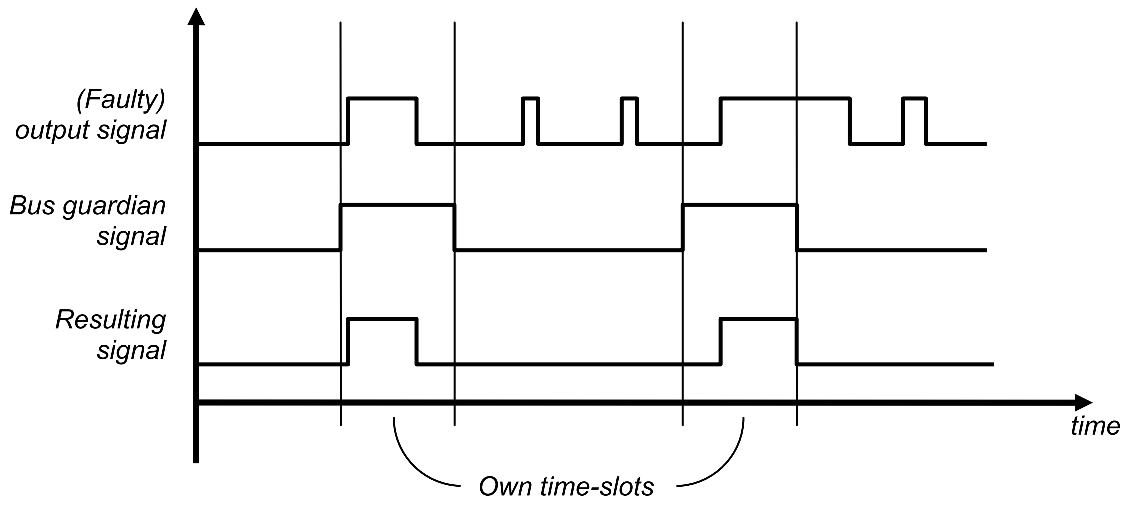

Figure 9 : Bus guardian effect on faulty output signal

Essentially, the bus guardian's signal is ANDed with the output signal of the communication handler of TEMPORAL APPLICATION DECOUPLING. Of course, partially truncated signals, where transmission overlaps own time slots, will not be received correctly by healthy nodes; therefore, the healthy nodes will recognise the erroneously sending node as being faulty.

5.5 Implementation

There are several implementations of BUS GUARDIAN possible. One solution is to realise it as some sort of filter, the other to AND its output with that of the communication handler through a simple AND-gate, as indicated in Figure 10.

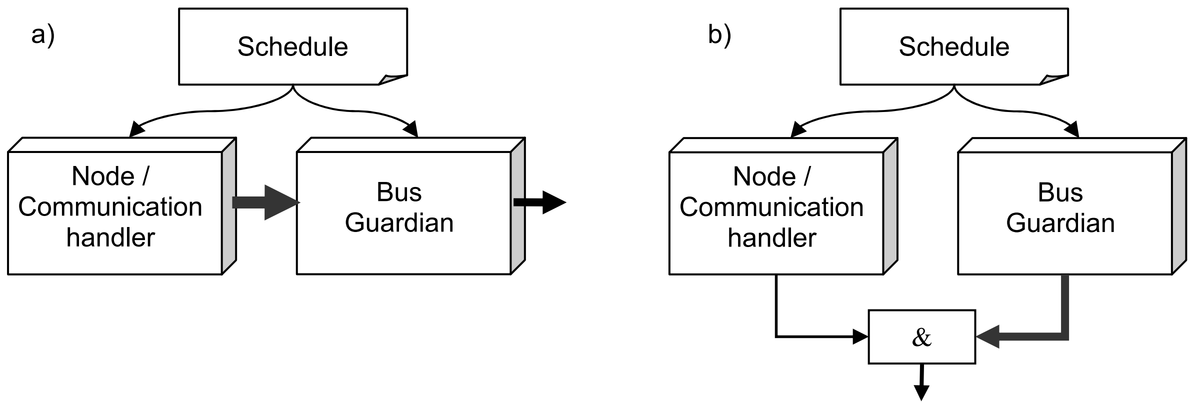

Figure 10 : Possible implementations of BUS GUARDIAN

A variant of alternative b) would be that the guardian itself controls a gate in the output line of the node communication handler.

An important aspect is the independence of the bus guardian from the element it has to guard. If, for instance, it shares physically the schedule presentation, internal clock, and power supply with the communication handler, and is even located on the same board close to it, then common mode failures are more likely to occur, diminishing the effectiveness of the BUS GUARDIAN pattern. Geographic separation, own power supply, duplicated schedule presentation, and a separate clock source, raise the implementation costs, though.

When a bus guardian fails, it may either not allow any transmission of its guarded node at all, which results in a fail-silent FCU containing the bus guardian and node guarded by it. Or it may allow transmission at wrong period phases. As long as the guarded node transmits correctly, its transmission will be corrupted, resulting in being considered as faulty by the remaining nodes. As a consequence, a faulty bus guardian will be recognized in the same way as a faulty node.

Alternative: Central BUS GUARDIAN

A possible alternative to individual bus guardians is to use a star configuration of the bus, and place the bus guardian into the centre of the star, where it opens the line for each node corresponding to the schedule. Then, only one bus guardian is needed for the whole network, which, however, turns it into a single point of failure. This can be neutralised by duplicating the whole network (which is necessary for fault-tolerance in any case). And since then only two bus guardian units are needed, they may be more expensive.

Bus guardians essentially consist of two functional components: the one which, according to the schedule, decides when to open and close the output channel of its guarded output unit, and the one which actually controls that channel. While the first component is usually is realised in software, the latter is in general realised in hardware, because it is much simpler, and its correctness can easier be proved.

BUS GUARDIAN adds an important aspect of fault-tolerance to PRESCHEDULED PERIODIC TRANSMISSION, and is easier realised when TEMPORAL APPLICATION DECOUPLING has been applied. They often apply TIME-TRIGGERED CLOCK SYNCHRONISATION by themselves to avoid common mode failure with their guarded host.

5.6 Running Example Resolved

In our EMB application, we will use the solution shown in Figure 10 b). All bus guardians share the schedule description with their guarded nodes, and are placed on the same board as the communication controller (to reduce development costs), but they apply TIME-TRIGGERED CLOCK SYNCHRONISATION independently from their guarded hosts. This avoids that any error occurring during clock computation within their host gets propagated to them, which would cause the bus guardian to allow transmission at wrong times.

5.7 Consequences

Benefits

Suppression of out-of-sync transmissions. Actually, without BUS GUARDIAN a time-triggered communication cannot be realised fault-tolerant.

Liabilities

Increased system costs. The usage of bus guardians raises the hardware costs of the whole system.

Increased complexity. Bus guardians add complexity to the system architecture, which affects design and development as well as maintenance. A possible counter-measure is the usage of centralised guardians.

5.8 Known Uses

ARINC 659 uses a pair of communication handlers (called BIUs – Bus Interface Units – there); each operated as bus guardian of the other. TTP uses a bus guardian with separated power supply and clock, but on the same chip as its communication controller, and also shares the MEDL with it. A star solution as described is also possible. Although not yet published in detail, FlexRay seems to implement BUS GUARDIAN quite similar to TTP, but shares some resources with the node. In SPIDER, bus guardians are separated from nodes and placed in a central hub or star coupler.

5.9 Related Patterns

BUS GUARDIAN often apply TIME-TRIGGERED CLOCK SYNCHRONISATION by themselves to avoid common mode failure with their guarded host. BUS GUARDIAN can be regarded as a temporal variant of *OUTPUT GUARD* [Saridakis 03].

6. Temporal Application Decoupling

Also known as: APPLICATION/COMMUNICATION BUFFER

6.1 Context

Hard real-time systems which have to be distributed for fault-tolerance reasons, and where PRESCHEDULED PERIODIC TRANSMSISSION is applied to guarantee that requested transmission times are not violated.

6.2 Problem

How can it be achieved that all application tasks adhere to the transmission schedule with respect to send data at the correct points in time, as well as being able to accept data in that moment when they arrive? Or, the other way round: how can it be avoided that application tasks disturb the transmission schedule by not sending/receiving at the correct schedule phases?

While such a periodic transmission provides a bunch of significant benefits for communication in dependable hard real-time systems, it lays a severe burden on the application tasks executed on the nodes of the distributed system: they have both to send and receive at precise points in time. Being a bit too early or too late (typically counted in micro- or even nanoseconds) for the corresponding I/O-operation, a task will definitely miss its time slot.

One could think performing each I/O-dependent task in an own thread, which is suspended at the corresponding position and reactivated when that operation has been performed. This approach, however, not only requires a potentially large number of threads to be handled, which may be hard if not impossible to realise in the main application area of the Triple-T patterns, namely embedded systems, it also may lead to poor CPU-utilisation and therefore increased risk of deadline violation. It may also introduce some level of indeterminism, which is undesirable in hard real-time applications.

6.3 Example

Consider our brake-by-wire application, where we have a couple of sensors and actuators. Each of them must be read (sensors) or written (actuators) at a specific point in time. Coordinating these timing constraints with those of the transmission schedule can become very challenging for the application.

6.4 Solution

Decouple the production and processing of data to be transmitted from the actual transmission by providing a buffer where application tasks put the data to be transmitted as soon as they are available, where received data are stored for reading by the application tasks. Further, a task (with highest priority), which is activated according to the transmission schedule, sends data when the corresponding time slots are encountered, and puts received data relevant for the tasks on its node into the buffer. Therefore, this pattern has the following participants (see Figure 11):

The **application task** is responsible for producing and putting data into the buffer to be transmitted <u>before</u> the corresponding time slot within the current period arrives. Likewise, it is responsible to read data from the buffer <u>after</u> they have been received.

The **communication handler** adheres strictly to the periodic transmission schedule. According to that, it puts data to be sent from the buffer to the bus, as it takes data received from the bus into it.

Note that for data transmitted in time-triggered mode, no queuing is necessary, because according to the system state semantics of such data, every new value written to the buffer overwrites the previous value of the same (state) variable. This, however, is not necessarily true for information transmitted in time slots reserved for event-triggered data; here, application tasks have to read received information before it is overwritten in the next period.

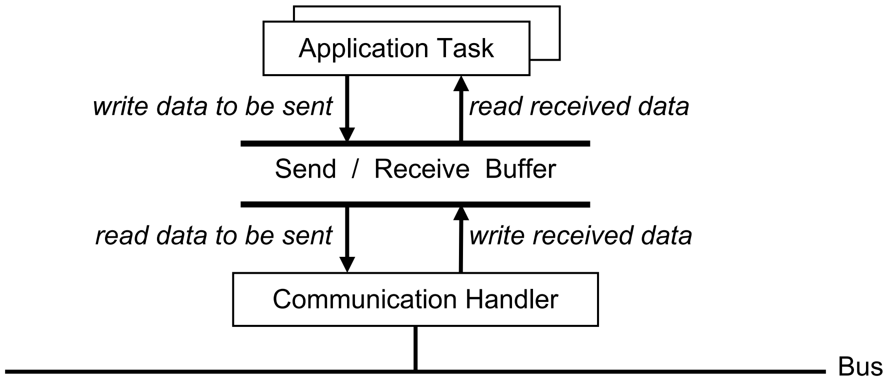

Figure 11 : TEMPORAL APPLICATION DECOUPLING structure

Figure 12 below shows an example of buffer access during a schedule period.

The activation boxes of the application tasks indicate that these do not necessarily start with reading and terminating with writing values, and are not necessarily executed strictly periodically. And the dashed arrow for "value w" indicates that data not needed on local host may but need not to be placed in the buffer.

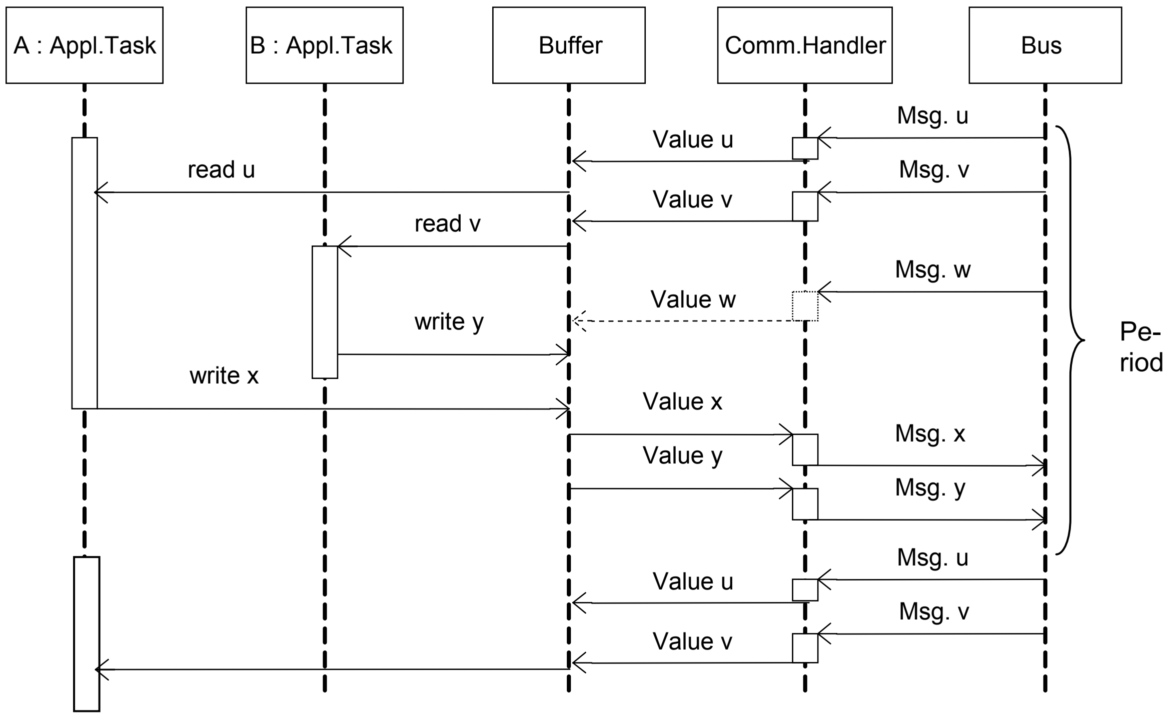

Figure 12 : Example sequence diagram for TEMPORAL APPLICATION DECOUPLING

6.5 Implementation

To allow simultaneous though controlled access to the buffer from both the application tasks and the communication handler, a dual-ported RAM could be used. The communication handler must have higher access priority than any application task.

Size and structure of the buffer can be derived from the transmission schedule. In its simplest form, it could have the same layout. For example, with respect to the schedule as sketched in Figure 4, it would start with an entry for message a, one for message b, and so on, thus making no distinction between entries for sending and receiving, respectively. Of course, no space is needed for messages received from other nodes, which are not relevant for application tasks on the local node. For them, no space needs to be reserved in the buffer.

Depending on the specific application, it may be necessary to assure that application tasks update their output fields in the buffer every time these fields have been sent, as they read and process their input fields every time these have been received. Typically, tasks have to write date to the buffer (and read them from it) with the same period they are transmitted. This can be checked by setting a flag for each buffer entry when it is written and reset it when it is read. Detecting a flag in the wrong state by the communication handler when accessing the corresponding buffer entry indicates a late application task, which can be interpreted as deadline violation and may trigger appropriate fault handling.

If executing the communication handler on the same CPU as the application tasks is too risky, it may be installed on another CPU.

6.6 Running Example Resolved

We will consider the EMB processing nodes only. These have to receive all sensor data, and provide their own output. Hence, their buffer could be arranged as shown in Figure 13.

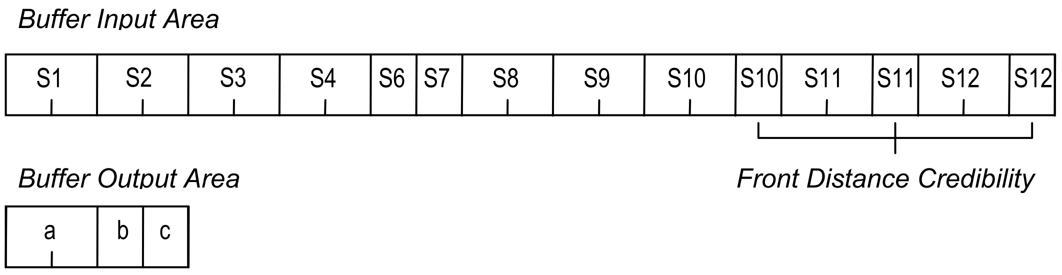

Figure 13 : layout of temporal application decoupling buffer of EMB PNs

The same message identifiers are used as in Figure 5, with the exception that PN output values are simply named 'a'…'c', because processing nodes do not care about the output of their replicas. For simplification, single bit values are represented by one byte in the buffer.

6.7 Consequences

Benefits

Simplification of application task scheduling. Equation constraints (i.e. "*operation x of task A has to be performed at time t*"), where 'x' denotes sending or receiving data, are replaced by range constraints ("*operation y of task A has to be performed between t_0 and t_1*"), where 'y' denotes writing data to the buffer or reading them from it, respectively.

Better CPU utilisation. Since there is more freedom for placing tasks along the time line, a more compact scheduling is usually possible.

Simple task monitoring. The read/write flags as mentioned in the Implementation clause can be used to easily detect deadline violations of application tasks.

Liabilities

Synchronised access on buffer needed. Race conditions have to be avoided where some buffer value is changed from one side while the other is reading it. If the hardware of the buffer memory does not directly support synchronised access, appropriate software mechanisms are not trivial.

Memory waste. The decoupling is done by memory cells (e.g. DPRAM). If memory is expensive, as it is usual the case for embedded systems, it must be kept as small as possible. FlexRay and TTCAN address this problem by storing only messages which are processed at this node.

6.8 Known Uses

In ARINC 659, a so-called *bus interface unit (BIU)* is used to decouple applications from the bus; in FlexRay *controller host interfaces (CHI)* for the same purpose; and TTP calls the communication handler *communication controller*, and the buffer *communication network interface (CNI)*. In TTCAN, a *Module Interface* is used for accessing Message RAM and Trigger Memory.

6.9 Related Patterns

The *HALF-SYNC/HALF-ASYNC* architectural pattern [Schmidt++00] resembles TEMPORAL APPLICATION DECOUPLING in that it separates synchronous from asynchronous processing layers. In contrast to *HALF-SYNC/HALF-ASYNC*, however, in TEMPORAL APPLICATION DECOUPLING the layering appears to be reverted: here, the lower layer (i.e. the communication system) is synchronous. In addition, TEMPORAL APPLICATION DECOUPLING does not use a queuing mechanism; instead, missing deadlines by the tasks within the upper, asynchronous layer is considered as failure.

If, for instance, tasks and communication handler are realised as threads within the same computing environment, the *synchronisation patterns* (e.g.*THREAD-SAFE INTERFACE*) as described in the same book can be used for controlling buffer access.

7. Acknowledgement

We owe a deep thank to our EuroPLoP'04 shepherd, Michael J. Pont. He helped us to add those details to turn the paper from being understandable mainly by people already familiar with the addressed topic into the current shape, which we hope is helpful to all who are interested in reliable communication in hard real-time systems.

We'd also like to express our gratitude to Arno Haase, Kevlin Henney, Kaspar von Gunten, Frank Buschmann, Juha Pärssinen, Andreas Rüping, Aimilia Tsanavari, Paris Avgeriou, and Asa MacWilliams, who workshopped this paper at the EuroPLoP'04 conference in a very constructive and encouraging though accurate way.

8. References

[Adams++96]	M. Adams, J.O. Coplien, R. Gamoke, R. Hanmer, F. Keeve, K. Nicodemus; *Fault-Tolerant Telecommunication System Patterns*; in J.Vlissides, J.O. Coplien, N.L. Kerth (eds.), Pattern Languages of Program Design 2 (PLOPD2), pp.549-562; Addison-Wesley, 1996; ISBN 0-201-89527-7
[ARINC 93]	Aeronautical Radio, Inc.; *ARINC Specification 659: Backplane Data Bus*; prepared by the Airlines Electronic Engineering Committee; Annapolis MD, Dec. 1993
[Cheng 02]	A.M.K. Cheng; *Real-Time Systems – Scheduling, Analysis, and Verification*; Wiley Interscience Series, Wiley & Sons, New Jersey 2002; ISBN 0471184063
[Driscoll++03]	K. Driscoll, B. Hall, H. Sivenkrona, P. Zumsteg; *Byzantine Fault Tolerance, from Theory to Reality*; in Proc. of SAFECOMP 2003, LNCS 2788, pp.235-248; Springer Berlin Heidelberg, 2003
[Douglass 03]	B. P. Douglass; *Real-Time Design Patterns – Robust Scalable Architecture for Real-Time Systems*; Addison-Wesley, 2003; ISBN 0-201-69956-7
[Hanmer++99]	R. Hanmer, G. Stymfal; *An Input and Output Pattern Language: Lessons From Telecommunications*; in N.B. Harrison, B. Foote, H. Rohnert (eds.), Pattern Languages of Program Design 4 (PLOPD4), pp.503-538; Addison-Wesley, 1999; ISBN 0-201-43304-4
[Führer++00]	Th. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther; *Time Triggered Communication on CAN (Time Triggered CAN – TTCAN);* in Proc. of 7th International CAN Conference 2000.
[ISO 03]	ISO 11898:2003 - Controller Area Network
[Kopetz++94]	H. Kopetz, G. Grünsteidl; *TTP – a protocol for fault-tolerant real-time systems*; in IEEE Computer, 27(1), pp.14-23; Jan. 1994
[Kopetz 97]	H. Kopetz; *Real-Time Systems – Design Principles for Distributed Embedded Applications*; Kluwer Academic Publishers, Mass. 1997 (2002); ISBN 0792398947
[Kopetz 03]	H. Kopetz; *Time Triggered Architecture;* ERCIM News No. 52, pp.24-25, January 2003
[Miner 00]	P.S. Miner; *Analysis of the Spider fault-tolerance protocols*; in C.M. Holloway (ed.), LFM 2000: 5th NASA Langley Formal Methods Workshop; Hampton, VA, June 2000

[Poledna++01] S. Poledna, W. Ettlmayr, M. Novak; *Communication Bus for Automotive Applications;* in Proc. of the 27th European Solid-State Circuits Conference; Villach, Austria, 18-20 September 2001

[Pont 01] M. J. Pont; *Patterns for Time-Triggered Embedded Systems – Building reliable applications with the 8051 family of microcontrollers*; Addison-Wesley, 2001; ISBN 0-201-33138-1

[Rushby 01] J. Rushby; *A Comparison of Bus Architectures for Safety-Critical Embedded Systems*; CSL Technical Report, Sep. 2001; SRI International

[Saridakis 02] T. Saridakis; *A System of Patterns for Fault Tolerance*; in A. O'Callaghan, J. Eckstein, Ch. Schwanninger (eds.), Proc. of the 7th EuroPLoP, 2002, pages 535-582; UVK, 2003; ISBN 3-87940-784-3

[Saridakis 03] T. Saridakis; *Design Patterns for Fault Containment*; in K. Hennley, D. Schütz (eds.), Proc. of the 8th EuroPLoP, 2003, pages 493-516; UVK, 2004, ISBN 3-87940-788-6

[Schmidt++00] Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.; *Pattern-Oriented Software Architecture 2 – Patterns for Concurrent and Networked Objects (POSA2)*; Wiley & Sons, 2000/2001; ISBN 0-471-60695-2

[Schoitsch++90] E. Schoitsch, E. Dittrich, S. Grasegger, D. Kropfitsch, A. Erb, P. Fritz, H. Kopp; *The ELEKTRA Testbed: Architecture for a Real-Time Test Environment for High Safety and Reliability Requirements*; in B. K. Daniels (ed.), Safety of Computer Control Systems, Proc. IFAC/IFIP symposium, 1990; Trondheim, London, Pergamon Press, 1991.

[Teepe++04] G. Teepe, F. Bogenberger; *FlexRay ist einsatzbereit;* Design & Verification, Februar 2004.

[Theuretzbacher 86] N. Theuretzbacher; *VOTRICS: Voting Triple-Modular Computing System*; FTCS-16, IEEE CS Press, Vienna, Austria, June 1986, (pp. 144-150).

[Welch++88] J. L. Welch, N. Lynch; *A new fault-tolerant algorithm for clock synchronization*; Information and Computation; 77(1), pages 1-36; April 1988

[Zeltwanger 04] H. Zeltwanger; *Time-triggered communication on CAN*, in Proc. of embedded world 2004 Conference, pp. 229-233, 2004