A Fault-Tolerance Layer for Distributed Fault-Tolerant Hard Real-Time Systems

Regular paper

C. Tanzer
TTTech Computertechnik GmbH
Schönbrunner Straße 7, A–1040 Vienna, Austria
phone: +43 (1) 876 62 36, fax: +43 (1) 877 66 92
e-mail: tanzer@swing.co.at

S. Poledna
TTTech Computertechnik GmbH

E. Dilger, T. Führer
Robert Bosch GmbH, Stuttgart

Abstract

This paper describes the conceptual model for, and the implementation of, a software fault-tolerance layer (FT-layer) for distributed fault-tolerant hard real-time systems. This FT-layer provides error detection capabilities, fault-tolerance mechanisms based on active replication, and the interface between the application software running on a node of the distributed system and the communication services. Communication is based on the fault-tolerant time-triggered protocol TTP/C. The FT-layer handles all necessary information transfer across the TTP/C bus transparently.

The conceptual model for the FT-layer is based on the DFR meta object model. This model is based on a separation of the three domains: *value domain, time domain,* and *distribution domain.* The DFR model supports a flexible choice of the degree of replication of fine-grained software components according to the application-specific dependability requirements. As the DFR model captures all the relevant design information explicitly, it allows the construction of powerful tools supporting the software development process.

One such tool, called xOLT, analyzes the application software and generates the FT-layer automatically and without user intervention. Due to a novel treatment of domain separation and system factorization, the FT-layer generated by the xOLT meets the stringent performance constraints of application areas extremely sensitive to cost such as automotive electronics.

Keywords: *fault-tolerance layer, time-triggered architecture, hard real-time, automatic software generation, distributed system*

1 Introduction

In the past, the deployment of computer technology by the automotive industry was mainly restricted to non safety-critical applications and to applications characterized by fail-safe behavior. Currently, there is a strong trend in the automotive industry towards an increasing number of safety related electronic systems for active and passive driver, passenger and environmental safety. These applications aim at increasing overall vehicle safety by freeing the driver from routine tasks and assisting the driver in critical situations. It is clear that this type of application requires computer systems providing high safety levels with fail-operational behavior.

The EU funded Brite-EuRam research project x-by-wire (Safety Related Fault-Tolerant Systems in Vehicles) addresses these requirements. Its objective is to develop a framework for the introduction of ultra-dependable electronic systems in vehicles which do not rely on conventional, i.e., hydraulic, mechanical, or electrical, backups. In addition, the requirement for high dependability must be met in the presence of severe cost constraints as imposed by the automotive industry.

In the x-by-wire project, the following system architecture has been chosen. The overall system consists of a set of clusters interconnected by gateways. Each cluster is a distributed system, comprising a set of processing nodes interconnected by the reliable communication service TTP/C [KoG94]. The fault hypothesis for a single node is fail-silence; the fault hypothesis for the communication service is omission failures, i.e., a message is either delivered correctly (with respect to both the value *and* time domain) or not at all.

The FT-layer has the task of separating the implementation of the non-functional requirements for fault-tolerance from the functional requirements as implemented by the application software. Thus, the FT-layer must handle error detection and error recovery transparently with respect to the application software [Lap92]. Furthermore, the FT-layer is responsible for handling the information transfer on the TTP/C bus with its replicated channels transparently. Figure 1 shows the rolê of the FT-layer in a layer model as a mediator between the application software on one side and the operating system and TTP/C communication network interface (CNI) on the other side.

The related literature describes a number of systems providing functions similar to the FT-layer. Examples are SIFT [WLG78], MAFT [KTW88], FTPP [HaL90], Delta-4 [CPR92], and GUARDS [WBB96]. However, these architectures are not applicable to extremely cost constrained markets such as automotive electronics. The reason for this is their high resource overhead in terms of the number of processors and processing cycles. Architectures such as SIFT, MAFT and FTPP are targeted for ultra high dependability in military applications. In these applications, the cost of system certification and validation is

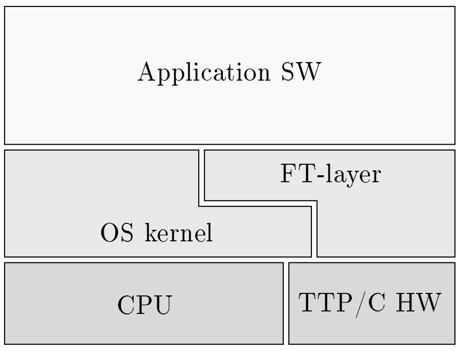

Figure 1: Rolê of FT-layer

orders of magnitude higher than the cost of the hardware components multiplied by the number of units shipped. Therefore, all these systems are built to tolerate Byzantine faults [PSL80]. The cost incurred by the necessary level of redundancy cannot be afforded in cost critical mass quantity produced applications. Furthermore, these architectures do not allow the selection of different error detection strategies and degrees of replication at the level of software components. Architectures such as Delta-4 and GUARDS are more flexible in respect to a fine grained selection of error detection strategies and replication levels. However, these architectures are again targeted at products where development cost outweighs product cost. The FT-layer approach described in this paper specifically addresses mass produced systems where product cost is extremely critical. The new approach described in this paper is based on the idea that best resource efficiency can be attained by generating the FT-layer with a tool which optimizes the performance of each application individually.

This paper is organized as follows: Section 2 presents the DFR model, which describes the application software architecture. Section 3 gives an overview of the underlying fault model for the FT-layer and it describes the strategies employed for node level error detection as well as the distributed fault-tolerance mechanisms. The actual implementation of the FT-layer together with its tool support xOLT is described in section 4. Finally, section 5 concludes the paper.

2 The DFR Model

The FT-layer is based on a meta object model designed to meet the specific requirements of distributed fault-tolerant hard real-time systems. This meta model defines a semantic framework which describes a software architecture for the construction of application specific object models. As such, it is also the ideal starting point for the construction of tools supporting the development of

DFR applications. This section gives a short overview of the DFR model; for a more detailed presentation the interested reader is referred to [PoT97].

The design of the DFR model was governed by the objectives ***composability, reusability, testability,*** and ***maintainability*** and by the goal of reconciling the constraints ***efficiency, guaranteed response time, robustness,*** and ***fault-tolerance***.

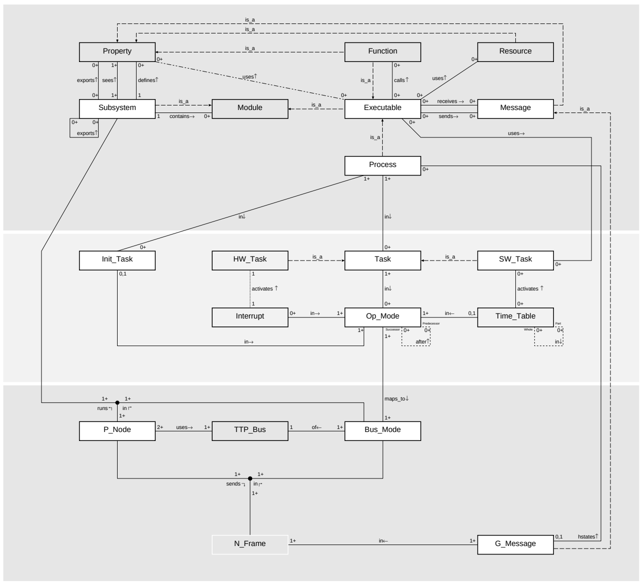

Figure 2: DFR meta object model

To achieve these objectives while satisfying the constraints, a separation of three different domains is necessary:

- ***Value domain:*** The value domain is concerned with the functional behavior of objects. It does not prescribe any aspect of synchronization or temporal events.
- ***Time domain:*** The time domain is concerned with the temporal behavior of the objects of a concurrent software system on a single node. The time domain addresses the dynamic interaction of independently active objects (of a node) and the events triggering changes in the system state.
- ***Distribution domain:*** The distribution domain is concerned with the association of software components to specific processing nodes in a distributed system.

These three domains are orthogonal to each other. The DFR model was designed in order to minimize the inter-dependencies between the objects in the different domains.

2.1 The value domain

The value domain is concerned with the functional behavior of objects. The central concept in this domain is the ***subsystem*** which encapsulates the functional characteristics of a real-time application object. The subsystem is the basic unit of packaging software components; it supports information hiding and hierarchical composition. In the DFR model, the subsystem is the unit of distribution, replication, and composability; it also is the focal point for the application of fault-tolerance and error-detection strategies.

A subsystem contains data and functions which are specified and implemented by:

- ***Processes.*** A Process is the smallest schedulable entity of a DFR system. It gives a *time-invariant* description of an independently active thread of control. Communication between processes is done exclusively with state messages (see below) and is anonymous, asynchronous, and non-blocking.
- ***Functions.*** Functions model algorithms; they execute in the context of the caller.
- ***Messages.*** Messages are used for communication between processes and have state semantics. State messages decouple the value domain very effectively from the time and distribution domains. Each process can use a message as if it were the only user — no concern about concurrency or distribution issues is necessary.
- ***Resources.*** Resources are used to guarantee mutually exclusive access to shared hardware devices. They are implemented with a stack-based priority ceiling protocol and thus completely avoid blocking, dead-locks, and live-locks.

The input of a process is defined as the set of messages received and the sensor readings performed by the process, if any. Its output is defined as the set of messages sent and the actuator commands given by the process, if any. The history state (h-state) of a process is the subset of the internal state retained between consecutive executions. The combination of process input and h-state is called input vector (i-vector). Correspondingly, process output and h-state are called output vector (o-vector).

2.2 The time domain

The time domain is concerned with the temporal behavior of the objects of a concurrent software system on a single node. The central concept in this domain is the ***operating mode*** (op-mode) which encapsulates the timing characteristics of a node-specific mode of operation.

An operating mode is defined by:

- ***Tasks***. Tasks link the time domain to the value domain by defining sets of processes to be executed in response to specific triggers. In the DFR model, the task is the unit of scheduling. A single task comprises a set of processes which are executed in sequential order.
- ***Time-table***. A time-table defines the time-triggered activation patterns of the tasks of an operating mode.
- ***Interrupts***. External hardware events are modeled as interrupts which activate event-triggered tasks.

The time domain is also concerned with the application of node-level error detection strategies.

2.3 The distribution domain

The distribution domain is concerned with the association of software components to specific processing nodes in a distributed system. The central concept of the distribution domain is the ***TTP-bus*** which connects a number of nodes called ***p-nodes***. A TTP-bus may operate in one or more ***bus-modes*** which are characterized by the transmission timing of global messages (***g-messages***).

Each p-node uses one or more TTP-busses to exchange messages with other nodes. A p-node connected to more than one TTP-bus works as a gateway node. In each bus-mode, a p-node executes a specific set of subsystems.

G-messages are used for the communication between nodes. The FT-layer maps them to (value domain) messages according to the distribution of subsystems across nodes.

The distribution domain is also concerned with the application of distributed fault-tolerance mechanisms.

2.4 Domain synthesis

The objects in the value, time, and distribution domains are as independent from each other as possible. This allows the software engineer to develop software components in isolation without having to consider the characteristics of the environment in which the components will be used.

The value domain is the fundamental domain and is independent from the two other domains. The time domain depends on the value domain, but is independent of the distribution domain — although it has to comply with the scheduling constraints of the distribution domain. The distribution domain depends on the interface of both the value and the time domain.

Due to this low inter-domain coupling, the components of the different domains can be developed independently of each other. Changes in one domain normally do not require changes in another domain. For instance, the same subsystem can be used unchanged in an isolated node and as part of a distributed system. The adaptation of the components to the actual environment is performed by the tool xOLT.

3 Error Detection and Fault-Tolerance Mechanisms

This section describes the mechanisms provided by the FT-layer for error detection and fault-tolerance. All the necessary code for these mechanisms is generated automatically by the tool xOLT. This tool based approach allows the selection of error detection strategies and redundancy levels for individual software subsystems. It is therefore possible to minimize the overhead for error detection and replication according to the dependability requirements of application functionalities.

3.1 Fault Hypothesis and Error Detection Strategies

The failure mode assumption for nodes is fail-silence. Each and every node must therefore deliver either results which are correct in both the value and the time domain or no results at all. It is therefore necessary to define the fault hypothesis for nodes. In the following, hardware as well as software faults are considered:

- Control timing faults.
- Control flow faults.
- Data flow faults.
- Data calculation faults.
- Data storage faults.

To achieve a sufficiently high coverage for the fail-silence assumption [Pow92], it is necessary to employ extensive error detection strategies at the node level to cope with the anticipated types of faults. In the following, we are concerned with software-based error detection strategies which can be applied systematically:[1] double execution, double execution with reference check, validity checks (for messages, history-states, and resources), assertion checking, and signature checks. Experimental results have indeed shown that it is possible to

[1]Systematically applicable means that the error detection strategy can be applied to a piece of software without any knowledge of the application domain [Pol96]. This for example excludes all types of plausibility and range checks.

achieve a high error detection coverage by combining these strategies [KFA95]. A brief description of the specific mechanisms and their implementations will be given in the following.

3.2 Double Execution

If double execution is specified for a given subsystem, all its processes are executed twice per activation. After the second execution a generic comparison function checks the individual o-vectors for bit-by-bit equivalence. The execution environment for processes with double execution looks like:

```
save i-vector
exec process
save o-vector-1
restore i-vector
exec process
save o-vector-2
compare o-vector-1 and o-vector-2
on error raise exception
send messages (generate output)
```

Double execution detects only transient hardware faults. However, the detection of transient faults is very important since their likelihood is orders of magnitude higher than the likelihood of permanent faults. This is particularly relevant for automotive electronics which must operate under harsh environmental conditions. Double execution can only detect transient faults when: (1) The persistence of the transient faults is short enough not to affect both executions, or (2) the transient fault affects both executions in different ways so that the o-vectors of the individual executions are different.

3.3 Double Execution with Reference Check

Since it is possible that a transient (or permanent) fault, such as a latch-up, affects both executions in the same way, it is desirable to detect this type of fault as well. This can be facilitated by performing an additional execution of the process with reference data between the first and second execution. That is, the process i-vector is initialized with a given set of reference data, the process is executed and its o-vector is compared to the results known to be correct for the given set of reference data. The code as generated by the tool xOLT looks like:

```
save original i-vector
exec process
```

```
save o-vector-1
i-vector := reference i-vector
exec process
compare o-vector and reference o-vector
on error raise exception
restore original i-vector
exec process
save o-vector-2
compare o-vector-1 and o-vector-2
on error raise exception
send messages (generate output)
```

By using reference data for the additional process execution, the correct result of the process execution is known *a priori* and can be checked. Since the additional process execution is inserted between the first and second execution with regular input data, it is very likely that any transient fault that affects the first and second process identically can be detected by the execution with reference data.

3.4 Message, Resource, and History State Validity Checks

Since information exchange between processes is carried out exclusively by means of message passing, it is important to detect message mutilations and timing faults. This is done by allocating additional memory for messages and by using error detecting codes. When sending a message, the message is transmitted along with the calculated error detection code. When receiving a message, the message data is read and the error detection code is calculated. Depending on the types of errors to detect, the error detection code is calculated over the actual message data, the message id, and a time-stamp. This allows detection of message data corruption, messages sent to wrong memory locations, and temporally invalid messages. Message validity checks are applicable to intra- as well as to inter-node communication.

Resources guarantee mutually exclusive access to shared memory or peripherals. In the DFR object model, a resource may protect a certain set of data. The coverage of the fail-silence assumption can be improved by using error detecting codes to detect mutilations of protected data.

The history state retains the information that must be kept by a process between two consecutive executions. To ensure that this information does not get corrupted between two executions, again, an error detecting code is used.

All the calls necessary to calculate and check the error detecting code are automatically inserted by the tool xOLT to relieve the application programmer from writing error detecting code.

3.5 Signature Checks

Signature checking is a technique to detect control flow errors [SaM90, MKG92]. The signature checking strategy implemented is a software based strategy applied at the level of subsystems. If a certain subsystem is selected for signature checks, then all the contained processes are subject to the signature check mechanisms. Accordingly, all tasks which contain processes of this subsystem perform the signature checking. Furthermore, all time-tables which contain the before mentioned tasks perform signature checking as well.

The signature checking mechanism is based on an error detecting code which is calculated over all the relevant task and process id's and checked against a predetermined value. This predetermined value is calculated off-line by the tool xOLT and represents a correct execution sequence. It is thus possible to detect omissions or superfluous executions of tasks and processes.

3.6 Assertion Framework

Assertions define predicates on the correctness of the system state which can be evaluated during run-time [Mey88]. There are three types of assertions: (1) *pre-conditions* are restrictions that must be fulfilled before a certain operation is executed, (2) *post-conditions* specify conditions which must hold after the execution of a certain operation, and (3) *invariants* are conditions that must hold throughout all legal executions of the system.

Within the software architecture used here, pre- and post-conditions are specified for processes. It is thus possible to check application specific correctness criteria before and after the execution of a process. If one of the predicates specified for the pre- or post-conditions is violated, then an exception is raised.

Invariants are specified for messages and resources. For messages, the predicate is checked before each receive operation and after each send operation. For resources, the predicate is checked before the get resource operation returns and after the release resource operation. This is necessary since read/write operations can be applied to the protected variables of a resource.

The predicates for processes, messages, and resources are specified for each object individually. The checking of these assertions, however, can be enabled and disabled on a per-subsystem basis in the time domain. Based on this information, the necessary code for the assertions is inserted automatically during system generation time by the tool xOLT.

3.7 Active Replication and Replica Determinism

With active replication, a specific subsystem executes on different nodes in parallel. All replicas receive the same inputs and generate the same outputs. If one node fails, the remaining replicas of the subsystem are able to continue their service on the remaining nodes. Examples of systems supporting active replication are SIFT [WLG78], MAFT [KTW88], and MARS [KDK89]. The "State Machine Approach", as described by Schneider [Sch90a], treats this replication method in a very detailed manner. For hard real-time systems, active replication is the technique best suited to mask faults transparently [Pol96], since there is no delay for recovery actions.

For active replication, it is necessary that all the replicated subsystems (which are executed on correct nodes) must exhibit *replica determinate* [Pol96] behavior, i.e., all contained processes produce identical results within a given time interval. There are basically two sources of non-determinism that must be addressed by the FT-layer: (1) readings of replicated sensors (this includes the value and time domain) and (2) differences in the execution order and speed. The mechanisms provided to handle these problems are described in the next two subsections.

3.8 Message Agreement

If a replicated subsystem sends a message that is based on replicated sensors it cannot be guaranteed that all the individual message instances are identical. It is therefore necessary to employ an agreement algorithm that returns one value. This is supported by the FT-layer and the xOLT based on the mechanism described in the following paragraphs.

Given a message m is sent with a replication degree of n. If the message is based on replica determinate information, all message instances m_1 to m_n are identical. According to the fault hypothesis, messages are delivered either correctly or not at all. The FT-layer can therefore present the first message instance m_i to the application software. We call the algorithm doing just this *pick first valid*.

However, if the message m is not replica determinate then some of its instances will have diverging values. The DFR model therefore allows the specification of an application specific agreement algorithm for the message in the distribution domain. Typical examples for such algorithms are *averaging agreement* or *majority voting*. During the generation of the FT-layer the xOLT applies the following strategy: If an application specified agreement algorithm is defined for a certain message, then this algorithm is used to read the message instances and to present one value to the application software. Otherwise, the default algorithm *pick first valid* is used.

It is important to note that user defined agreement algorithms are specified in the distribution domain. This guarantees that all nodes in the cluster use the same algorithm to handle messages.

3.9 Timed Messages

Actively replicated subsystems can show replica non-deterministic behavior as a result of slight divergences in the execution timing. To avoid node internal non-determinism it is necessary to achieve total agreement on the order of message receive and send operations. A very efficient means to attain this are *timed messages* [Pol97]. It has been shown for real-time systems that timed messages can guarantee ordered message delivery without any communication—except for external events.

A brief description of the timed message mechanism will be given in the following; for a more detailed discussion the interested reader is referred to [Pol96a, PBB97]. Real-time systems are characterized by the *a priori* knowledge of the time at which a certain process must finish. It therefore follows that individual processors have common knowledge of the completion times of replicated tasks. This information can be exploited to achieve ordering of the message send and receive operations. It should be noted that this scheme is necessary only for messages which are sent node-locally; for global messages the communication service already guarantees global order.

Upon sending a message, the message is marked with the completion time of the sender process. Since only local messages are considered here, messages arrive immediately after the completion of the send operation. It is therefore guaranteed that every message sent by a task arrives within the task's deadline. The associated task's completion time is called a message's *validity time*.[2] Typically, messages are sent more than once during system operation, i.e., each send operation generates a new message version. Message receive operations for timed messages are defined as follows: if a task receives a message, it needs to select the message version for which the associated validity time is the latest one before the receiver's task activation request time. This enforces replica determinism on the send/receive behavior of replicated processes.

3.10 Recovery and re-integration of nodes

After a node detects a fault, it must perform a self-test to determine whether the fault is transient or permanent. After a transient fault, it is advantageous to re-synchronize the node and re-integrate it into the cluster. After a permanent fault, the node leaves the cluster; in some cases it may be replaced by the re-integration of a new, correct node into the cluster.

The FT-layer handles node recovery completely autonomously. If the FT-layer determines that a node starts into an already running cluster[3], then a re-integration is performed. During re-integration, the FT-layer receives all messages and h-states on the bus. This information is stored to form valid and

[2]In this context, validity time means *not to use before* rather than *not to use after* time.
[3]This information is provided by the TTP/C communication protocol.

up-to-date i-vectors for the processes. Having done this for a sufficient period (one cluster cycle, c.f. subsection 4.2), the application software is started in synchronization with the cluster. This ensures replica determinism of the newly integrated node. The prerequisite requirement for re-integration—that the h-state of replicated processes is sent on the bus—is guaranteed by the tool xOLT which generates the FT-layer accordingly.

4 The FT-Layer Implementation

The FT-layer provides the interface between the application software running on a single p-node and the TTP/C bus connected to the node. In terms of the DFR model, the FT-layer connects the value and time domains with the distribution domain. In particular, the FT-layer handles the sending and receiving of global messages and their mapping to node-local messages. For the application software, the distribution domain is irrelevant and there is no difference between a local and a global message.

The FT-layer is generated by the software tool xOLT. This tool is an extension of the off-line tool OLT which was developed as a part of the ERCOS operating system. ERCOS is a commercially available product which is used in several industrial automotive projects.

4.1 xOLT

The off-line tool OLT was developed to support the development of robust, reusable, and composable software components for embedded hard real-time systems with rigorous resource constraints.

The development of the OLT was governed by the goals:

- Full support for the DFR model.
- Decoupling of software components.
- Decoupling of the application software from the used infrastructure, e.g., compiler, operating system, target hardware, networking, etc.
- Declarative instead of imperative interface.
- Automatic consistency checks, including user-defined checks.
- Extensive off-line optimization.

The OLT performs message allocation, resource management, code adaption, code optimization, and consistency checks. The original implementation handles the value and time domain, i.e., it supports the development of applications on isolated embedded processors. The xOLT extends the OLT by supporting the distribution domain; in particular, it performs the automatic generation of the FT-layer. The xOLT works on a per-node basis, i.e., it knows the value and time domains of a single node and the distribution domain of the complete distributed system.

The OLT and xOLT perform two major functions: ***analysis*** and ***synthesis***.

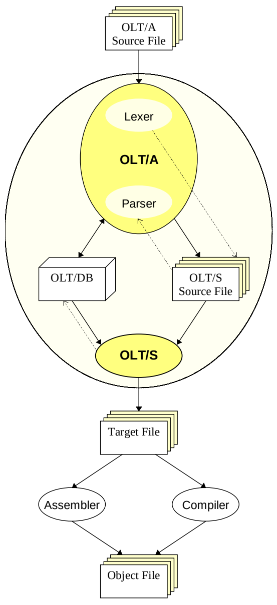

Figure 3: OLT structure

The analysis part (OLT/A) gathers information about the application objects and their associations. The source files read by OLT/A are written in a host programming language like C or assembler augmented by OLT declarations. These OLT declarations provide all the design information required by the DFR model. The OLT/A extracts the DFR specific information from the source files and puts it into an internal database (OLT/DB). The source files can be analyzed in any sequence and at any time — all together or one after another. The OLT/DB contains all necessary information about the global system structure needed by the OLT.

The synthesis part (OLT/S) uses the information in the OLT/DB to synthesize a complete application. Synthesis is possible only if the OLT/DB is complete and consistent. The OLT/S performs the functions:

- Global consistency checks.
- Message allocation and optimization.
- Resource handling and optimization.

- Stack space allocation for the different priority and exception levels.
- Time-table resolution.
- Configuration of the ERCOS kernel.
- Source code rewriting: adaption of reusable components to the application environment and to the target system.

The OLT supports a powerful macro language which provides full access to the information in the OLT/DB. This macro language is used extensively for the implementation of the FT-layer.

4.2 Communication basics

A TTP/C network comprises a set of p-nodes connected by the two replicated channels of a TTP/C bus and is called a ***cluster***. Access to the bus is controlled by a cyclic time-division multiple access (TDMA) schema derived from a global concept of time. The sequence of slots in which each p-node sends at most once forms a TDMA ***round***. After the completion of one TDMA round, the next round commences with the same temporal access pattern, but possibly with different messages. The number of different rounds determines the length of the ***cluster cycle***. After a cluster cycle is finished, the transmission pattern starts all over again at the beginning of the next cluster cycle.

A p-node consists of two components: the host computer and the TTP/C controller(s). The TTP/C controller handles all access to the TTP/C bus autonomously and communicates with the host computer via the communication network interface (CNI).

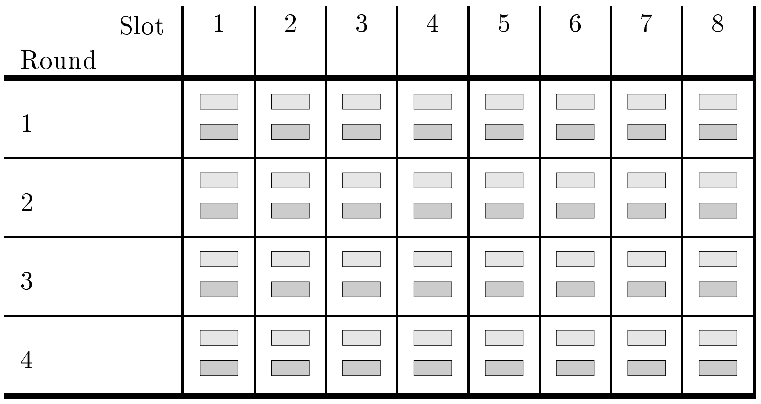

Figure 4: Slots vs. Frames

For each slot, a specific p-node is allowed access to the TTP/C bus. The p-node sends one frame on each of the channels. Each of these frames may be an ***i-frame*** (used for protocol management) or a ***n-frame*** (used for message transfer between hosts). Each n-frame can carry between one and sixteen g-messages. A g-message might be sent in one or both n-frames of a slot depending on how critical the message is.

A non-multiplexed p-node has access to the same slot in every round of the cluster cycle. A multiplexed p-node shares the same slot with other multiplexed p-nodes, each of them using the slot in a different round of the cluster cycle.

4.3 Replica handling

Replication of a subsystem means that there are n replicas of the subsystem running on different p-nodes of the cluster. A message of a replicated subsystem is sent in as many slots per round as there are replicas, i.e., if one replica sends the message in a round, all others have to send it, too. Each replica may send the message in one or two n-frames per round. The users of the replicated message don't need to know about the replication degree.

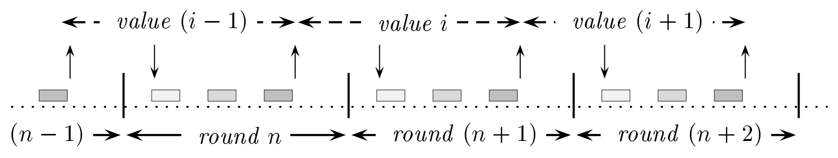

Figure 5: Replica Handling

A replicated message can be read from the CNI only after the slot of the last replica was processed by the TTP/C controller. This situation is shown in diagram 5 for three replicas (the slots of all other p-nodes are omitted). Between the first and the last of the replica's slots, the old value of the message must be used. Only after the arrival of all slots can the value be updated (marked by the up-arrow).

A replicated message must be written into the CNI before the transmission of the slot of the first replica commences (marked by the down-arrow). That means that all replicas must update the CNI at the same time. Otherwise, replica determinism might be compromised.

No matter what the replication degree of a message, there is at most one slot per round where the FT-layer must read or write the message to or from the CNI, respectively. For replicated messages, only the last slot in the round is used for reading the message from the CNI; only the first slot is used for writing the message into the CNI.

For replica-deterministic messages, the value carried by any valid n-frame can be used. Due to the fail-silent assumption, the values of all frames should contain the same value.

For non-replica-deterministic messages, the FT-layer must produce a replica-deterministic agreement on the values of all p-nodes from which a valid n-frame was received. The agreement algorithm used is specified per g-message, i.e., different g-messages may use different agreement algorithms. In the value domain, only the replica determinism status of a message is specified. There isn't any difference in the application's code between the sending/receiving of a deterministic and a non-deterministic message.

4.4 FT-Layer generation

The xOLT generates the tasks implementing the FT-layer by using the information in the OLT/DB. The design of the FT-layer generation was governed by the objectives:

- Complete transparency to the application.

  The application programmer needs no knowledge about the TTP protocol, the CNI, or the bus schedule.
- Avoid unnecessary work.

  Only messages actually used by the application (or the FT-layer itself) are handled.
- Avoid unnecessary tasks and unnecessary time-table entries.

  By combining the handling of the different messages into as few tasks as possible, unnecessary overhead for the operating system kernel is avoided.

For messages used by event-triggered tasks, every slot carrying the message must be processed. As the activation time of an event-triggered task is not known in advance, every slot must be considered in order to avoid the use of stale information by the task. Thus, event-triggered tasks impose an overhead on the FT-layer.

For messages used only by time-triggered tasks, all receive-slots which are not followed by a task using the message are ignored. It makes no sense to process a slot when the value received is not used before the next slot containing the message arrives. Diagram 6 shows an example: white slots are receiving slots, gray slots are sending slots. For slot 1, both messages are used and must be handled; slot 2 can be ignored because none of its messages is used; for slot 4, only the message m1 is used and must be handled; for slot 5, the messages m3 and m4 are not used and can be ignored. Send-slots are always written, even if there was not any change in the value of the message.

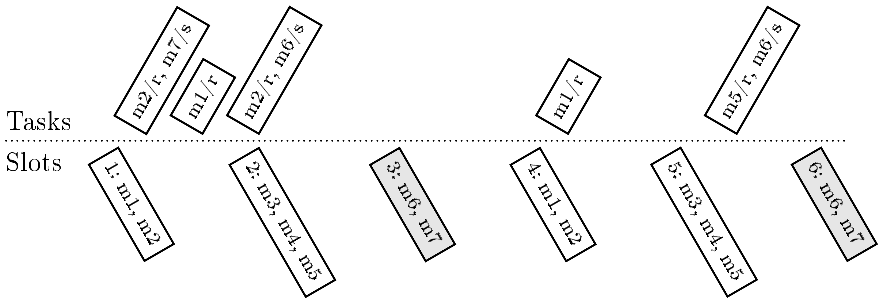

Figure 6: Slot Pruning

The processing of a slot can be shifted in the interval between slot transmission and task activation. For received messages, the earliest time of slot processing is given by the completion of slot transmission, the latest time is given by the time of task activation (for time-triggered tasks). For sent messages, the earliest time is that of task termination (task activation plus deadline), the latest time is given by the start of slot transmission minus the deadline of the FTL task. That means, for every slot we have an interval for scheduling the FTL task (this interval collapses to a point for messages used by event-triggered tasks). This interval can be used for optimization.

Messages with overlapping processing intervals are handled by the same FTL task. Each receive task is scheduled as early as possible, each send task is scheduled as late as possible. The FTL tasks inherit the priority of the tasks using the messages. This guarantees that a FTL task cannot be interrupted by an application task using the same message.

According to the strategy just outlined, the xOLT augments the time domain with FT-Tasks and additional time table entries. This is done automatically without any user intervention. For the generation of the FT-layer, the xOLT needs complete information about the global system structure (in particular, about the message use). This information is only available after the analysis of all source files. The FTL objects are defined via the normal OLT mechanisms, i.e., they must be also analyzed by the OLT.

The time domain is defined by a single source file. To generate the FT-layer, the xOLT analyzes this file twice — the first run determines the timing of the application specific tasks, the second run adds the tasks generated by the FT-layer via the OLT macro language and enters them into the time-tables of all affected op-modes. Because the FTL objects are defined by normal OLT mechanisms, their use is handled by exactly the same mechanisms as all application specific objects (e.g., message copying).

Any change in the use of global messages (i.e., a change of a source file defining objects of the value domain) automatically triggers a re-analysis of the time domain. This keeps the FT-Layer consistent at all times with the application software.

5 Conclusions

This paper presents a conceptual model for, and the implementation of, a software fault-tolerance layer (FT-layer) for distributed fault-tolerant hard real-time systems. This FT-layer is based on a systematic, application-independent framework for fault-tolerance supported by the code-generating tool xOLT. The FT framework comprises node-level mechanisms for increased error detection coverage and distributed mechanisms for tolerating permanent faults of processing nodes and peripherals.

The presented architecture supports a clear separation between the implementation of application-specific, functional requirements and the implementation of non-functional requirements for systematic fault-tolerance. This allows the tool xOLT to generate the FT-layer completely automatically and thus supports the development of reusable software components. In addition, the xOLT substantially increases the productivity of the application programmer.

By using knowledge of the global system design, the xOLT is able to perform extensive optimizations and therefore to generate highly efficient code for the FT-layer (and for the application-specific code). In contrast to the overhead imposed by generic mechanisms without tool support, the xOLT generates code custom-tailored to the properties of a specific application.

In the context of the x-by-wire project, the xOLT was developed by extending the existing commercially available tool OLT. The implementation of the xOLT has been finished and the tool is being used to develop the software for the x-by-wire prototype application. First results indicate an excellent performance of the tool and the generated FT-layer.

References

[CPR92] Chèréque, M., Powell, D., Reynier, P., Richier, J.-L., Voiron, J.: 1992, *Active Replication in Delta-4*; In Proceedings of the 22th International Symposium on Fault-Tolerant Computing (FTCS-22), IEEE Computer Society Press, Boston, Massachusetts, Jul. 8–10, 1992, pp. 28–37.

[HaL90] Harper, R.E., Lala, J.H.: 1990, *Fault-Tolerant Parallel Processor*; Journal of Guidance, Control and Dynamics, Vol. 14, No. 3, May–June 1990, pp. 554–563.

[KDK89] Kopetz, K., Damm, A., Koza, C., Mulazzani, M., Senft, C., Zainlinger, R.: 1989, *The MARS Approach*;
IEEE Micro, Vol. 9, No. 1, Feb. 1989, pp. 25–40.

[KFA95] Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G.: 1995, *Integration and Comparision of Three Physical Fault Injection Techniques*; in [RLK95, pp. 309–327].

[KTW88] Kieckhafer, R.M., Thambidurai, P.M., Walter, C.J., Finn, A.M.: 1988, *The MAFT Architecture for Distributed Fault-Tolerance*; IEEE Transactions on Computers, Vol. 37, No. 4, 1988, pp. 394–405.

[KoG94] Kopetz, H., Grünsteidl, G.: 1994, *TTP*, A Protocol for Fault-Tolerant Real-Time Systems; IEEE Computer, Vol. 27, No. 1, Jan. 1994, pp. 14–23.

[Lap92] Laprie, J.C. (ed).: 1992, *Dependability*, Basic Concepts and Terminology; Volume 5 of Dependable Computing and Fault-Tolerant Systems, Springer-Verlag New York, Inc.

[MKG92] Miremadi, G., Karlsson, J., Gunneflo, U., Torin, J.: 1992, *Two Software Techniques for On-Line Error Detection*; In 22th International Symposium on Fault-Tolerant Computing, 1992, pp. 328–335.

[Mey88] Meyer, B.: 1988, *Object-Oriented Software Construction*; Prentice Hall Book Co., Inc., Englewood Cliffs, New Jersey. ISBN 0-13-629031-0.

[PBB97] Poledna, S., Barrett, P., Burns, A., Wellings, A.: 1997, *Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems*; Submitted for publication to IEEE Transactions on Computers, 1997.

[PSL80] Pease, M., Shostak, R., Lamport, L.: 1980, *Reaching Agreement in the Presence of Faults*; Journal of the ACM, Vol. 26, No. 2, Apr. 1980, pp. 228–234.

[PoT97] Poledna, S., Tanzer, C.: 1997, *DFR Objects*, A Meta Object Model for Distributed Fault-Tolerant Hard Real-Time Systems; Submitted for publication to *ISORC '98, The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.*

[Pol96] Poledna, S.: 1996, *Fault-Tolerant Real-Time Systems*, The Problem of Replica Determinism; Kluwer Academic Publishers. ISBN 0-7923-9657-X.

[Pol96a] Poledna, S.: 1996, *Optimizing Interprocess Communication for Embedded Real-Time Systems*; In Proceedings of the Real-Time Systems Symposium. Washington, 1996, pp. 311–320.

[Pol97] Poledna, S.: 1997, *Deterministic Operation of Dissimilar Replicated Task Sets in Fault-Tolerant Distributed Real-Time Systems*; In Proccedings of the Sixth IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-6). Grainau, Germany. March 1997. Springer Lecture Notes Series.

[Pow92] Powell, S.: 1992, *Developing in C++ Using Dynamic Link Libraries*; Object Magazine, Vol. 1, No. 5, January/February 1992, pp. 50–57.

[RLK95] Randell, B., Laprie, J.-C., Kopetz, H., Littlewood , B., (eds).: 1995, *Predictably Dependable Computing Systems*; Springer-Verlag New York, Inc.

[SaM90] Saxena, N.R., McCluskey, E.J.: 1990, *Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums*; IEEE Transactions on Computers, Vol. 39, No. 4, April 1990, pp. 554–559.

[Sch90a] Schneider, F.B.: 1990, *Implementing Fault-Tolerant Services Using the State Machine Approach*, A Tutorial; ACM Computing Surveys, Vol. 22, No. 4, Dec. 1990, pp. 299–319.

[WBB96] Wellings, A., Beus-Dukic, L., Burns, A., Powell, D.: 1996, *Genericity and Upgradability in Ultra-Dependable Real-Time Architectures*; In Work in Progress Proceedings, Real-Time Systems Symposium, Dec. 1996, pp. 15–18.

[WLG78] Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostack, R.E., Weinstock, C.B. : 1978, *SIFT*, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control; In Proceedings of the IEEE, Vol. 66, No. 10, Oct. 1978, pp. 1240–1255.

Contents