DIPLOMARBEIT

A Framework for Validation of Virtual Gateways in the DECOS Architecture

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines/r

Diplom - Ingenieurs/Diplom - Ingenieurin

unter der Leitung von

o. Univ. - Prof. Dr. Hermann Kopetz

und

Univ. Ass. Dipl. - Ing. Dr. Roman Obermaisser

als verantwortlich mitwirkendem Assistenten
Institut für Technische Informatik 182/1

eingereicht an der

Technischen Universität Wien,
Fakultät für Informatik

durch

Bakk. Techn. Christian Pucher
Matr. - Nr. 0125353
Kirchenstraße 42, A–2225 Loidesthal

Wien, im Dezember 2007 .....

Abstract

The DECOS integrated architecture tries to combine the benefits of federated and integrated architectures by decomposing a large real-time system into nearly-independent distributed application subsystems (DASs) and integrating them on a single distributed computer system.
The communication infrastructure for the DASs is provided by virtual networks that are implemented on top of a time-triggered core architecture. Often it is necessary for a DAS to exchange information with another DAS, which is the task of a gateway. Gateways in the DECOS architecture are implemented as hidden gateways (i. e. transparent to the application). Gateways not only have to forward messages from one network to another, they are also used to resolve operational or semantic property mismatches between the interconnected networks. Another important task of gateways is the provision of an encapsulation service that is responsible for error containment and the selective redirection of messages in order to save bandwidth.
The goal of this thesis is the validation of the gateway services by experimental evaluation. To solve this task in a convenient way, we proposed the use of a measurement framework that has the ability to monitor the system under test and is capable of automated execution of testruns.
First we show a model of a framework that is generally applicable. The model consists of a test controller and monitor and an execution environment in which the tests are executed. The test controller feeds the execution environment with parameters that are extracted from test descriptions and controls and monitors the target hard- and software at runtime. We pursue the approach of software implemented fault injection at runtime. Faults are injected by providing faulty parameters to the application.
Then we present an actual implementation for the evaluation of our exemplary DECOS cluster. Test descriptions are written in XML and are read by the control application that extracts the necessary parameters for the test runs and provides them to the target system. The application jobs on the target system are responsible for the execution of testruns and recording their operations which are then analyzed by the control application. The properties we are interested in are latencies and bandwidths via the gateways as well as the correct functionality of their error containment and selective redirection mechanisms.

Kurzfassung

Die DECOS Architektur ist eine integrierte Architektur und versucht, die Vorteile von verteilten und integrierten Architekturen zu vereinen. Dazu wird das Gesamtsystem in nahezu unabhängige Teilsysteme, sogenannte Distributed Application Subsystems (DASs), unterteilt.
Die Kommunikationsinfrastruktur für die DASs wird von virtuellen Netzwerken zur Verfügung gestellt, welche auf einer zeitgesteuerten Basisarchitektur implementiert sind. Oft ist es notwendig für ein DAS, Information mit einem anderen DAS auszutauschen, dies ist die Aufgabe eines Gateways. Gateways sind in der DECOS Architektur als virtuelle versteckte Gateways implementiert (d.h. unsichtbar für die Applikation). Neben dem Weiterleiten von Nachrichten von einem Netzwerk ins andere, sind Gateways auch dafür verantwortlich, etwaige semantische oder operative Unstimmigkeiten zwischen den verbundenen Netzwerken aufzulösen. Eine weitere wichtige Aufgabe von Gateways ist das Encapsulation Service, welches für die Fehlereingrenzung verantwortlich ist, sowie für die selektive Weiterleitung von Nachrichten, um Bandbreite zu sparen.
Ziel dieser Arbeit ist die Validierung der Gateway Services durch experimentelle Evaluierung. Um diese Aufgabe bequem zu lösen, stellen wir ein Framework vor, das die Möglichkeit bietet, das System zur Laufzeit zu überwachen und welches automatische Testläufe unterstützt.
Zuerst präsentieren wir ein Modell, welches allgemein anwendbar ist. Es besteht aus einem Testcontoller und -monitor und dem Zielsystem. Der Testcontroller gewinnt aus Testbeschreibungen die Parameter für die Testläufe und steuert und überwacht das Zielsystem während der Laufzeit. Wir verfolgen den Ansatz der software-implementierten Fehlereinstreuung. Fehler werden dadurch ins Zielsystem eingestreut, indem wir der Zielanwendung fehlerhafte Parameter zur Verfügung stellen.
Danach gehen wir auf die Implementierung eines konkreten Frameworks für unseren Beispiel DECOS Cluster ein. Testbeschreibungen im XML Format werden von der Kontrollanwendung gelesen, welche die notwendigen Parameter für die Testläufe extrahiert und sie dem Zielsystem zur Verfügugng stellt. Die Anwendungen auf dem Zielsystem sind für die Ausführung der Testläufe, sowie für das Aufzeichen der Operationen verantwortlich, welche im Anschluss an einen Test von der Kontrollanwendung ausgewertet werden. Die Eigenschaften, die uns dabei am meisten interessieren sind die minimalen, maximalen und durchschnittlichen Laufzeiten und Bandbreiten über die Gateways, sowie die korrekte Funktionalität deren Fehlereingrenzungsmechanismen.

Contents

Chapter 1

Introduction

Large distributed safety-critical real-time systems like they are in use in the automotive and avionics domain are often implemented as federated systems. In a federated system, every application subsystem has its own dedicated computer system that is only loosely coupled to other subsystems. The main advantage of this approach is the inherence of fault containment, i. e. it is very unlikely that a fault in one application subsystem influences the function of another subsystem. A big drawback, on the other hand, are the high costs due to the need for a dedicated computer system for every new application subsystem.

An alternative approach is the use of an integrated architecture where several application subsystems are integrated within a single distributed computer system. Obviously, integrated systems reduce hardware costs as well as costs regarding wiring, space, weight, cooling and maintenance.

An architecture that tries to combine the benefits of both, the federated and the integrated approach, is the DECOS integrated architecture [OPT07] where the real-time computer system is decomposed into a set of nearly-independent subsystems called distributed application subsystems (DASs), each of which provides a part of the of the overall system's functionality. The DASs are distributed across several components, interconnected by a time-triggered core network.

On top of the physical core network, virtual networks [OPK05b] are implemented as overlay networks that provide the communication infrastructure for the distributed application subsystems. Due to the fact that DASs may need to communicate among one another, gateways are needed to interconnect virtual networks. Gateways in the DECOS architecture [OPK05a] do not only serve the purpose of forwarding messages from one virtual network to another but also provide selective redirection of messages and error containment between subsystems. The encapsulation service of virtual networks and gateways is a prerequisite for spatial and temporal partitioning [Rus99] between applications in an integrated system which

is inherent in federated systems.

1.1 Objective of the Thesis

The main purpose of this thesis is to show that the gateways of a DECOS cluster ensure spatial and temporal partitioning in the presence of faults.

For the purpose of experimental evaluation of the gateways in our prototype implementation, we identified the need for a measurement framework that supports automated testing and easy parametrization to offer a wide variety of test scenarios. The framework has to fulfill the following requirements:

Monitoring of the system under test without probe effects: This means, that the exertion of measurements must not affect the intended function of the system in the temporal or value domain.

The ability to induce faults into the system: In our framework, we want to induce controlled (i. e. with a specific timing) software faults resulting in a *timing message failure* or a *naming message failure*.

Reproducibility of tests: This requirement claims the ability of an experiment to be accurately reproduced by someone else or some time else.

In our solution, we pursue the approach of a measurement framework that accepts testcase specifications written in XML which describe the communication behavior of the target applications. The parameters are extracted from the specifications and during the startup phase of the cluster, they are written to a shared memory that provides a point-to-point communication between a server and the application jobs. Faults are injected into the system by providing faulty parameters to the application. The target application jobs are also responsible for data acquisition for subsequent analysis of the executed tests.

The measurements gathered by the target applications are transferred to the server after an experiment is finished and not sent during its execution. This prevents the earlier mentioned probe effect.

1.2 Structure of the Thesis

In chapter 2 we give an overview of projects that are closely related to the context of this thesis.

Chapter 3 is dedicated to the concepts that are used throughout this thesis. We begin with a comparison of federated and integrated architectures and some examples for integrated architectures in section 3.1. Section 3.2 introduces the DECOS integrated architecture which is the chosen architecture for our validation framework. In section 3.3, we compare the event-triggered and the time-triggered control paradigm. The fundamental concepts of dependability are described in section 3.4. Section 3.6 explains partitioning mechanisms that are crucial for fault containment in an integrated architecture. The remainder of this chapter is dedicated to virtual networks and gateways that are important for the integration and interconnection of multiple subsystems on a single platform.

Chapter 4 describes the model of a validation framework with automated test procedures. Section 4.1 starts with the logical structuring of the DECOS architecture into virtual networks and jobs, interconnected by gateways. In section 4.5, we describe a model for automated test procedures. In section 4.3 we distinguish software and hardware faults and section 4.5.1 introduces fault injection techniques that are used in the automated tests.

The actual implementation of the measurement framework is the topic of chapter 5. We begin with the description of the hardware and software setup of the prototype cluster in section 5.1 and section 5.2. The implementation details of the framework are subject of section 5.3.

Chapter 6 is dedicated to the experimental evaluation of of the implementation. We begin with the hypothesis for the experiments in section 6.1 followed by a set of test cases in section 6.2. In section 6.3, we discuss the results of the experiments and finish with a concluding interpretation of those results.

We finish with a conclusion in chapter 7 that summarizes this theses and proposes the future work.

Chapter 2

Related Work

2.1 Assessment of Error Detection Mechanisms in the Time-Triggered Architecture

Ademaj's thesis [Ade03] is dedicated to the evaluation of error detection mechanisms in the Time-Triggered Architecture [Kop98]. Fault injection is introduced as a valuable methodology for the validation of computer systems.

Ademaj uses software fault injection to emulate physical faults in the components of a TTP [TTT02a] node.

2.1.1 Hardware Setup

The cluster setup for Ademaj's validation framework consists of four active TTP nodes, a passive node and a PC [Ade03].

- The **fault injection node** (FI-node) is the node exposed to fault injection
- The **golden node** is an exact replica of the FI-node and produces the same outpust results as the FI-node in a fault-free run.
- The **comperator node** compares the messages of the FI-node and the golden node and detects fail-silent violations in the value domain.

The fourth active node is required for the correct function of the clock synchronization algorithm. The passive node does not send messages on the bus, instead it forwards all messages transmitted from the active nodes to a PC. It is also used for reconfiguration of the cluster.

2.1.2 Software Setup

The software setup for the SWIFI environment comprises the following parts [Ade03]:

- Fault injection code for the TTP/C controller that is executed along with the protocol code.
- Fault injection code for the host controller that is executed along with a host application or the FTcom layer tasks.
- Monitoring code for the host controller.
- Fault injection manager that is responsible for providing fault injection data and triggering fault injection.

2.2 Fault Injection Experiments Using FIAT

FIAT (Fault Injection-Based Automated Testing) is a tool for the experimental validation of a system's dependability. Barton et al. [BCSS90] review the experimental validation environment of FIAT and show that software fault injection is *a viable method to test and validate systems* [BCSS90]. FIAT uses the following abstractions for the fault injection process:

A **workload** is an observable set of real-time tasks that are communicating via observable communication channels. The fault injector is linked together with the task code of the workload.

A **fault class** describes a set of workload modifications that represent a group of logical or physical faults with common properties.

Data collection is the process of recording the events during a test. FIAT distinguishes between *history* (the normal behavior of the system) and *error reports* (exceptions and abnormal events).

Experiments are transformed by an Experiment Description Translator (EDT) into an experiment script that contains a command sequence for controlling fault injection at runtime.

The FIAT hardware consists of two types of components, namely the fault injection manager (FIM) and the fault injection receptor (FIRE).

The **fault injection manager (FIM)** supports the experiment development and data collection/analysis. FIM is also responsible for runtime control of an experiment.

The **fault injection receptor (FIRE)** provides the execution environment for the experiments and is controlled by commands from the FIM based on the experiment descriptions.

2.3 NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors

NFTAPE [SFKI00] represents a fault injection framework that composes automated test fault injection experiments from other readily available lightweight fault injectors. According to the authors, *no single tool is sufficient for injecting all fault models.*

The key features of NFTAPE are identified as follows:

- Multiple Fault Models
- Multiple Fault Triggers
- Multiple Targets
- Versatile Error Reporting Methods

Stott's paper [SFKI00] discribes two actual implementations of an NFTAPE framework, one uses a hardware fault injector with a Myrnet LAN while the other one uses a SWIFI fault injector.

2.4 Comparison to this thesis

The framework presented in this thesis can best be compared to the FIAT fault injector [BCSS90] because we also use a control application, comparable to the FIM, that generates the parameters for the experiments and controls the execution environment. One significant difference is the user application running on the execution environment which doesn't interact directly with the control application but uses parameters provided by the control application at startup.

The most important difference is our integrated architecture which integrates several application subsystems within a single distributed computer system. Since we use a time-triggered core architecture, we can make use of the global time base which allows us to inject faults at predefined points in time.

When we compare our framework to NFTAPE [SFKI00], the most important commonality is the support for multiple fault models, in our case naming message failures and timing message failures.

Chapter 3

Concepts

This chapter begins with an introduction to integrated systems and a short overview about the DECOS integrated architecture as it is the underlying architecture for our validation framework. The following sections explain some fundamental concepts that are used throughout this thesis. Section 3.3 compares event-triggered and time-triggered systems and section 3.4 explains the fundamental concepts of dependability. The last sections of this chapter introduce the mechanisms that are crucial for the implementation of an integrated system, those are partitioning (cf. section 3.6), virtual networks (section 3.7) and gateways (section 3.8).

3.1 Federated vs. Integrated Systems

In the context of real-time systems for distributed applications, we can distinguish between two classes of systems, *federated* and *integrated systems*. In a federated system each application subsystem has its own dedicated computer system which is only loosely coupled to other application subsystems. Federated systems have been preferred for ultra-dependable systems, because fault containment is inherent.

Integrated Systems are characterized by integrating several application subsystems in a single distributed computer system with internal replication to provide fault tolerance. Since a single component in an integrated system hosts functions of multiple application subsystems, there is a potential to diminish fault containment between those functions. Therefore, an integrated system must provide partitioning to ensure protection against fault propagation among functions [OPT07].

This paper will focus on the DECOS integrated architecture [OPT07] which will be described in detail in the following section 3.2. *AUTOSAR* [AUT07], *Integrated Modular Avionics (IMA)* [Rus99] and *PAVE PILLAR* [OSB+87] are other

examples for integrated architectures.

IMA is a distributed real-time system that is in use aboard civil as well as military aircraft. It is capable to host multiple applications of differing criticality levels on a single component. Furthermore, IMA is intended to make technology transparent to the applications [Rus99].

PAVE PILLAR was developed for military aircraft, especially for advanced tactical fighters. It allows to implement the core avionics tasks, Digital Signal Processing, Mission Processing, Vehicle Management Processing, and Avionics Systems Control, with common hardware and computer programs [OSB+87].

AUTOSAR (Automotive Open System Architecture) [AUT07] is an example for an integrated architecture for the automotive industry. Its key features are the definition of a modular software architecture for electronic control units (ECUs), the standardization of interfaces and a runtime environment that provides inter- and intra-ECU communication across all nodes of a vehicle network [AUT07].

3.2 The DECOS Integrated Architecture

The DECOS (Dependable Embedded Components and Systems) integrated architecture [OPT07] provides a framework for the development of distributed embedded real-time systems. It combines the benefits of federated and integrated systems by subdividing the real-time system into nearly-independent distributed application subsystems (DASs) with different criticality levels and different requirements. Those DASs are implemented on top of a time-triggered core architecture that supports the safety requirements of the highest considered criticality class. According to [OPT07], the core architecture for safety-critical real-time systems must provide the following services:

- deterministic and timely transport of messages
- fault-tolerant clock synchronization
- strong fault isolation
- consistent diagnosis of failing nodes

DASs are grouped into a safety-critical subsystem and into a non safety-critical subsystem. The DAS-specific high-level services (Virtual Network Service, Encapsulation Service, etc.) form an abstraction and are intended to hide the implementation of the core architecture from the applications. Details about the logical structure of the DECOS architecture are described in section 4.1.

3.3 Event-Triggered vs. Time-Triggered Architectures

Temporal control signals for the activation of a task can arise from two different sources [Kop97]:

1. If the control signal is derived from a significant state change (called an event) in the environment or within the system, the system is called an event-triggered system. An event can be the arrival of a message at a node or an external interrupt like a pushed button.

2. A system where control signals are derived from the progression of real-time is called time-triggered system. The temporal control signal is generated whenever the real-time clock reaches a priori determined global points in time, specified in the scheduling table.

The different nodes of a distributed real-time computer system are interconnected via a real-time communication network. Within a node, the host computer is connected to the communication controller via a *communication network interface* (CNI) [Kop97]. There are two different types of messages that are exchanged via the communication system, namely *event messages* and *state messages*.

Event messages combine event semantics with external control [Kop97]. They require exactly-once processing and synchronization between sender and receiver to prevent loss of information.

State messages combine state-value semantics with autonomous control [Kop97]. There is no need for synchronization between sender and receiver, because state variables are updated in place. The CNI is usually implemented as a dual-ported RAM and no control signals need to cross the CNI.

3.3.1 Event-Triggered Communication

Event-triggered (ET) communication systems are designed for the transport of event messages. *The temporal control is external to the communication system* [Kop97], it is triggered by the host computer or by the arrival of a request message from another node. Temporal control in an ET system depends on the behavior of all nodes in the system and therefore they are not composable.

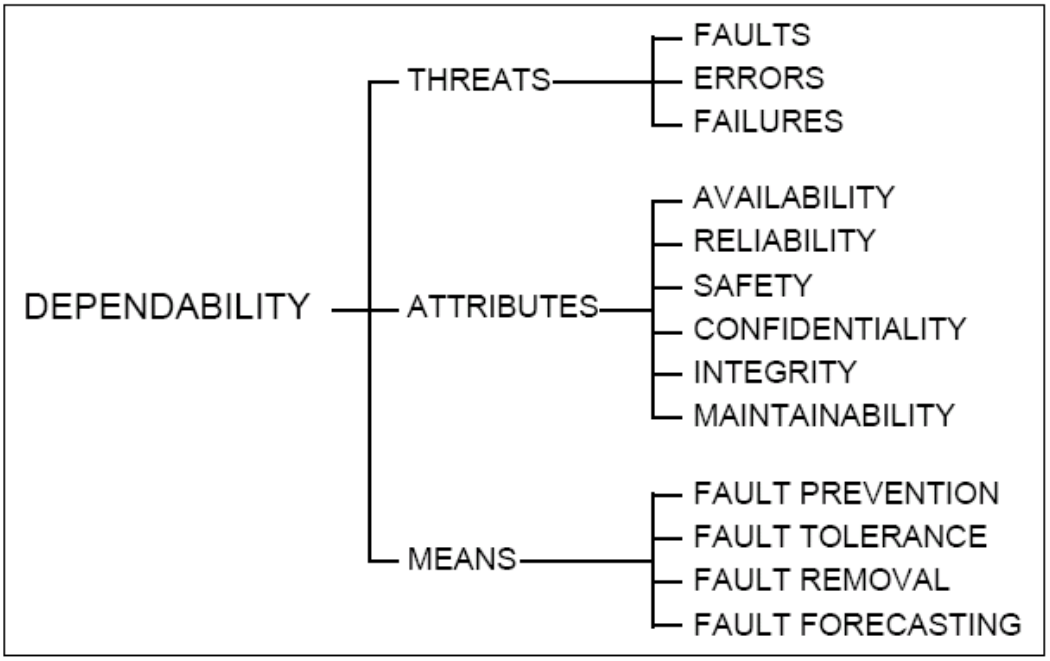


Figure 3.1: The dependability tree [ALR01]

3.3.2 Time-Triggered Communication

In at time-triggered (TT) communication system, temporal control resides within the communication system [Kop97]. It is not dependent on the behavior of the host computers of the nodes. The communication system autonomously transports state messages from the sender's output port to the receiver's input port at a priori determined global points in time, and no control signals cross the ports. The CNI acts as *temporal firewall* [KN97], *isolating the temporal behavior of the host computer from the temporal behavior of the communication system* [Kop97].

3.4 Dependability

The Definition of dependability of a computing system according to [ALR01] is *the ability to deliver a service that can be justifiably trusted.* Correct service is delivered, when the service implements the system's function (i. e. what the system is intended to do). Dependability consists of three parts, namely the threats to, the attributes of and the means by which dependability is attained. This characterization is shown in the *dependability tree* [ALR01] in figure 3.1.

3.4.1 Attributes

Availability means dependability with respect to readiness for usage. It is a measure of the delivery of correct service with respect to the alternation of correct and incorrect service. Availability is measured by the fraction of time that the system is ready to provide service.

Reliability quantifies the continuity of service. It indicates the probability that a system provides a specified service until a given time.

Safety means *absence of catastrophic consequences on the user and the environment* [ALR01]. Kopetz [Kop97] differentiates between *malign* (critical) failure modes and *benign* (noncritical) failure modes. A failure mode is said to be malign if the cost of a failure can be orders of a magnitude higher than the utility of the system during normal operation [Kop97].

Confidentiality stands for the *absence of unauthorized disclosure of information* [ALR01].

Integrity is intended to avoid incorrect system state modifications and is a prerequisite for availability, reliability and safety.

Maintainability describes the ability of a system to be repaired and modified. It is a measure of the time needed to repair a system after the occurrence of a failure [Kop97].

3.4.2 Means

Fault prevention is intended to prevent the occurrence of faults by employing quality control mechanisms during design and manufacturing phases of a hardware or software product. Those mechanisms can be design rules for hardware and structured programming or modularization for software.

Fault tolerance is the ability of a system to deliver correct service in the presence of faults. This is achieved by error detection and system recovery. Error detection can be concurrent (during service delivery) or preemptive (while service delivery is suspended). System recovery means the transition from an erroneous system state i. e. a state that contains one or more errors to an error-free system state. Error handling can be obtained in three ways: *rollback* returns the system to a saved state called checkpoint, *compensation* eliminates errors by making use of redundancy and *rollforward* carries the system to a new state without errors. In order to prevent faults from being activated again, four steps have to be followed: *fault diagnosis* to locate and typify the fault, *fault isolation* for logical exclusion of faulty components, *system reconfiguration* (e. g. activating spare components) and *system reinitialization*.

Fault removal is intended to reduce faults during the development phase and the operational phase of a system. Validation is a technique to check the system behavior against its specification. If any deviation from the intended system behavior is revealed, fault diagnosis and corrective mechanisms are performed to prevent the fault from being activated again.

Fault forecasting evaluates the system behavior with respect to occurrence or possible activation of faults. This evaluation process can either be qualitative (ordinal) or quantitative (probabilistic).

3.4.3 Threats

A system is said to have *failed* if the service perceived by the user deviates from the specified service. A *failure* is the consequence of an unintended system state called *error*. The physical or logical cause of an error is called fault. A fault is active if it produces an error and is dormant otherwise [ALR01] [Kop97].

Faults

Faults are the physical or logical causes of errors, and thus the indirect cause of a subsequent failure. They can be categorized into *Fault Nature* (by chance or intentional), *Fault Perception*, *Fault Boundaries* (internal or external), *Fault Origin* (development or operation) and *Fault Persistence* (permanent, intermittent or transient) [Kop97].

Transient faults occur at a particular point in time, remain for some time and then disappear. Possible reasons for transient faults are temperature, pressure, UV light, electromagnetic interference, etc. Faults that occur from time to time are called *intermittent faults*. They are caused for example by loose interconnections or aging of components. *Permanent faults* occur at a particular point in time and remain until the faulty component is repaired. A possible example is a defective IC board due to the manufacturing process (solder splashes) or a design fault [Ade03].

Errors

An error is an incorrect, an thus unintended system state. If an error exists only for some time and then disappears, it is called a *transient error*. If the error persists until the erroneous component is repaired, it is called a *permanent error* [Kop97].

Transient errors are typical for systems, where operation can be characterized by periodic duty cycles (e. g. control loops). An error in the previous duty cycle can not affect the result of the current or subsequent duty cycles. An example for a

system prone to permanent errors is a database application where the corruption of one data element can influence the future system behavior.

Failures

Kopetz [Kop97] defines a failure as *an event that denotes a deviation between the actual service and the specified or intended service.* Failures can be classified by *Failure Nature* (value or timing), *Failure Perception* (consistent or inconsistent), *Failure Effect* (benign or malign) and by *Failure Oftenness* (permanent, intermittent or transient).

A *value failure* occurs when the value of a delivered service does not comply with the specification. A *timing failure* means that a service is delivered too early, too late, or not at all (infinitely late) [Ade03].

Regarding the perception of a failure, we can distinguish between *consistent failures* where all users see the same (possibly wrong) result and *inconsistent failures.* In an inconsistent failure scenario, the malicious subsystem can present contradictory behavior to each of the correctly operating subsystems. They are also called *two-faced failures*, *malicious failures* or *Byzantine failures* [Kop97].

3.5 Fault Injection

Faults are rare events that occur infrequently during system operation but in order to evaluate the behavior of a system in the presence of faults, fault occurrence has to be accelerated. A technique to insert artificial faults into the system is called fault injection. Ademaj identifies two purposes of fault injection:

> **Verification** checks *if the system behaves as specified in the presence of faults* [Ade03].
>
> **Validation** is a technique to produce *experimental data about the likely dependability of fault-tolerant systems* [Ade03].

Fault injection can be classified into software implemented fault injection and hardware fault injection.

3.5.1 Software Implemented Fault Injection

Software implemented fault injection (SWIFI) is performed by a piece of code in order to emulate faults in locations which are accessible to software (memory

locations and registers). Its main advantages are the high controllability in the space and time domain and the reproducibility of fault injection experiments. Faults can be injected either pre-runtime or during runtime:

Pre-runtime SWIFI: When a program instruction is modified before it is executed on the target hardware we talk about pre-runtime SWIFI. The fault is activated when the modified part of the program is executed.

Runtime SWIFI requires additional code that is added to the target software and is executed along with the target program. There are several ways to trigger the injection of faults. One way is to use a time-out mechanism (timer interrupt) to specify the time when a fault is activated. Fault injection can also be triggered by external hardware exceptions (interrupts) or by the use of a software trap instruction before a particular program instruction. When the fault injector is implemented as part of the user program, we talk about *code insertion*.

3.5.2 Hardware Fault Injection

Hardware fault injection causes stress of the target system by inserting physical perturbations like an electromagnetic field or radiation. Hardware injection techniques can be classified into two categories, depending on the way how faults are injected into the system:

Hardware fault injection with contact: In this case, the fault injector has direct physical contact with the target system. An example of hardware fault injection with contact is *pin-level fault injection* which alters the logical level of signals at the pins of an IC.

Hardware fault injection without contact: The fault injector has no direct physical contact with the target system. Examples of this technique are radiation based fault injection, electro-magnetic interference (EMI) fault injection and laser based fault injection.

3.6 Partitioning

The main purpose of partitioning is fault isolation. As mentioned in the previous section, in an integrated system a single physical component is shared by multiple functions. To ensure that a fault in one function does not propagate to

another function, each function has its own dedicated partition. Partitioning is not intended to protect against a hardware failure of a component and therefore, all functions located on a physical component have a high probability of common mode failures.

Without partitioning, a failure of a low-criticality function could cause a function of a higher criticality class to fail which means that all functions have to be assured to the level of the most critical function. Partitioning eliminates that need and allows every function to be assured independently to its criticality class.

Rushby [Rus99] distinguishes two dimensions of partitioning, namely spatial and temporal partitioning which will be described in detail in the next paragraphs.

3.6.1 Spatial Partitioning

Spatial partitioning has to ensure that software in one partition cannot corrupt software in another partition nor corrupts the use of private devices of other partitions [Rus99].

Spatial Partitioning within a single processor

Spatial Partitioning is concerning the possibility that software of one partition overwrites memory of other partitions. Modern processors usually use a *Memory Management Unit* (MMU) to avoid violations of spatial partitioning. Memory accesses from user applications are translated by the use of tables. The OS kernel is responsible for managing those MMU tables and providing each partition with disjoint memory locations. The kernel also uses the MMU mechanisms to protect itself from being corrupted by user software. The kernel is furthermore responsible for arranging context switches between partitions so that a partition can be suspended and resume execution at a later point in time. Rushby [Rus99] distinguishes two types of partition swapping arrangements, referred to as *restoration* and *restart* models, respectively.

The wide functionality of MMUs of today's processors usually exceeds the requirements for embedded systems, instead a simple, fixed allocation scheme would be more adequate.

Another technique to ensure spatial partitioning is *Software Fault Isolation* (SFI) [WLAG93]. This method performs array bounds checking not only on references indexed into an array but on all memory references. This is done by checking machine code statically or during runtime for instructions using register interdict addressing mode prior to its use.

To minimize pathways for fault propagation it is important to restrict inter-

partition communications to those that are needed. Since inter-partition communication is usually asynchronous it is important to take care when dealing with time-sensitive data. There is a possibility when reading a sample from another partition that the value is already out of date (temporally inaccurate). Kopetz proposes a solution called *temporal firewall* , where an *accuracy interval* is provided along with state-data that tells that the sample is *accurate until* the indicated time [KN97].

Communication also takes place between partitions and devices. Devices have to be protected against access by the wrong partition, they must not violate the partitioning scheme and they may be partitioned themselves. When a partition has exclusive access to a device, memory mapped I/O is the usual way and MMU mechanisms are sufficient. If a device is shared among partitions, there are two cases. The first case where different partitions have read access to a device is not problematic. In the second case where more than one partition has write access to a device, things are more complicated. In this case a device management partition is needed to coordinate accesses to the device.

3.6.2 Temporal Partitioning

Temporal partitioning has to ensure that software in one partition does not corrupt the temporal properties of another partition, including performance, latency, jitter and the duration to access its service [Rus99].

Temporal Partitioning within a single processor

In the context of real-time embedded systems it is important not only that results are correct but they are also delivered at the right time.

The worst concern is that faulty software in one partition might monopolize the CPU, crash the whole component or it might issue a dangerous instruction like a HALT and thus denying service to all partitions running on the same component. System crashes are usually prevented by spatial partitioning mechanisms and instructions that can halt the cpu are not allowed in user mode but there are a number of bugs reported where user-mode instructions can halt the CPU when supplied with certain parameters [SPL95].

The usual way to deal with overrunning tasks is taking control away from the affected partition by some kind of timeout mechanism. This will ensure each partition to get enough access to the CPU but in real-time systems this is not sufficient. Real-time tasks need to be executed at the right time with predictable latencies and jitter.

Real-time systems can be scheduled statically or dynamically. In a dynamically

scheduled systems the points in time when a task is executed is decided at runtime. This is usually done by assigning a fixed priority to each task. Priorities have to be chosen in a way that the overall system behavior is predictable and that all tasks satisfy their deadlines. The most common algorithm for dynamic scheduling is rate monotonic scheduling (RMS) [LL73]. In a statically scheduled system, tasks are executed cyclically at a fixed rate. The schedule is calculated during system development according to the maximum execution time of every task. The advantage of static scheduling is the complete predictability of the system behavior and the simplicity of its implementation. The drawback of this scheduling scheme is the wasteful handling of sporadic tasks (since they are statically scheduled too) and that long-running tasks have to be split up because of a more frequent task.

3.6.3 Partitioning in a Distributed System

Components in a distributed system usually share a communication medium, such as a bus. When communicating between partitions on two different processors the receiving partition can either be scheduled dynamically or statically. While the dynamic case is less complicated, a static schedule requires interrupts to be latched until next execution of the particular partition or that partition is guaranteed to execute at the arrival of interrupts. A useful way to avoid these consequences is the use of a concentrator device that buffers incoming data without imposing load on the CPU.

Babbling idiot failures are failures where one transmitter constantly sends (random) data and thus denying service to other transmitters by monopolizing the bus. Rushby [Rus99] distinguishes between *babbling partitions* and *babbling processors*. A babbling partition can be controlled by a global scheduling scheme. A babbling processor has to be prevented at the transmitter by the use of a mediator component which has to be aware of the communication schedule and allows the transmitting component to access the bus only at specific points in time according to the schedule. It is important that the mediating component is guaranteed to fail independently.

Globally scheduled time-triggered systems guarantee that the bus is free when a component is supposed to transmit but locally scheduled event-triggered systems have to deal with contention between components attempting to access the bus at the same time. In CAN (Controller Area Network) [Bos99] for example a priority based arbitration scheme is used to resolve contention. Such systems only provide probabilistic bus-access delays and are weak in the presence of faults.

3.7 Virtual Networks

A virtual network is an overlay network on top of a physical core network. In the DECOS architecture, virtual networks are implemented on top of a time-triggered core network. As described in section 3.2, the distributed real-time system is subdivided into distributed application subsystems, in order to achieve the benefits from both, federated and integrated system architecture. Each of those DASs is provided a dedicated virtual network that fits the requirements of the specific DAS.

3.7.1 Characterization of Virtual Networks

A virtual network provides the communication infrastructure of a DAS and can be characterized by the following properties [OPK05b]:

Functionality: The transport of messages is the basic functionality that a virtual network has to provide. Different *communication topologies* like point-to-point or broadcasting can be distinguished. The functionality may encompass additional services such as flow control [OPK05b].

Operational Properties: The operational properties describe the characteristics in the value and temporal domain. The temporal properties specify available bandwidths, transmission latencies, and the employed control paradigm (time-triggered or event-triggered). The syntactic properties describe message format used in a specific virtual network.

Namespace: Message names are either explicit (part of message syntax) or implicit (defined by send and receive instants) and are defined according to the respective DAS. In order to preserve existing namespaces of legacy applications and to achieve the ability of independent development of DASs, virtual networks provide a separate namespace to each DAS [OPK05b].

Dependability: A virtual network can be described by the dependability properties discussed in section 3.4. Those properties arise from the underlying core architecture and from the implementation of the virtual network's high-level services.

3.7.2 Time-Triggered Virtual Networks

Considering properties like predictability, error detection, fault tolerance and replica determinism, the time-triggered control paradigm is better applicable [Kop95], [Rus01]. On this account, safety-critical DASs are strictly time-triggered. Time-triggered virtual networks can also be employed for a non-safety critical DAS if state messages are appropriate.

The sender pushes information into the memory element of its output port, while the receiver pulls information from the corresponding input port. The virtual network autonomously transports state messages from the sender's output port to the receiver's input port at a priori determined global points in time, and no control signals cross the ports. This is the concept of a *temporal firewall* [KN97] which prevents temporal fault propagation by design. Due to the fact that no acknowledgement messages are necessary and therefore no back pressure is exerted on the sender, we talk about *implicit flow control*.

3.7.3 Event-Triggered Virtual Networks

Event-triggered virtual networks are designed for the sporadic exchange of event messages. In order to maintain state synchronization between sender and receiver, exactly-once processing is inevitable. Therefore, messages have to be buffered in message queues at the input and output ports.

Event-triggered virtual networks can support *implicit* and *explicit flow control*. Explicit flow control occurs when acknowledgement messages are sent by the receiver and thus back pressure is exerted on the sender. The transmission of a message through the sender depends on control flow in the opposite direction [OPK05b].

3.8 Gateways

In an integrated system as introduced in [OPT07], communication takes place not only within a subsystem but also between subsystems. In order to couple those subsystems, gateways are required. Transparently forwarding information is the minimum function of a gateway [GZ79]. In the DECOS architecture, three types of gateways are discussed, namely *hidden gateways*, *virtual gateways* and *physical gateways*.

> A *hidden gateway* is a gateway at the architectural level. It is transparent to jobs at the application level.

A *virtual gateway* is a gateway within a cluster. It connects two virtual networks within a cluster. Virtual gateways not only have to forward messages from one network to another, they need to resolve discrepancies in the operational specifications of two *distributed application subsystems* (DASs) [OPT07].

A *physical gateway* connects the physical networks of two different clusters. It can be situated within a distributed application subsystem (DAS) or between two DASs.

Gateways are used to resolve differences in the operational specifications (*property mismatches*) of two different DASs. If for example an event-triggered DAS and a time-triggered DAS are connected, there are obvious differences in the temporal specifications.

3.8.1 Property Mismatches

According to Jones, a *property mismatch* is a *disagreement among connected interfaces in one or more of their properties* [J+02]. If such a mismatch is not resolved, a failure can occur during system operation.

When dealing with virtual gateways that are built on top of the same physical network, property mismatches occur only at the operational and semantic levels [OPK05a].

Semantic Level

We talk about a semantic mismatch, if the receiver of a message interprets its meaning differently from the meaning originally intended by the sender of the message.

A particular case of a semantic mismatch is incoherent naming, where the same name is assigned to different entities in two different subsystems. Consider for example two CAN networks linked together by a gateway, where the same identifier can refer to a temperature value in one network and an engine control value in the other.

Another case of a semantic mismatch between two DASs occurs when one subsystem uses messages with state semantics, while the other one uses event semantics.

Operational Level

If there are differences in the operational specifications of ports and links, we talk about an operational property mismatch. Those mismatches can be of syntactic nature (e. g. different data types or message lengths) or can arise from different temporal specifications of the interconnected networks. In the latter case, this happens when connecting a time-triggered with an event-triggered network or when two time-triggered networks operate with different periods or phase-shift relationships [OPK05a].

3.8.2 Property Transformation

In order to resolve property mismatches as described in the previous section 3.8.1, the gateway performs transformations on the information exchanged via the gateway.

Semantic Level

Property mismatches at the semantic level are usually resolved by an application-level gateway instead of a generic gateway because the mismatches are typically highly application specific [OPK05a].

In the integrated architecture, incoherent naming is resolved by providing a different name space to each DAS. If a message has different names in different DASs, it is necessary for the gateway to change the message name before forwarding the message to the target network.

Operational Level

Operational mismatches are resolved through generic hidden gateways. In order to perform syntactic property transformations, the gateway requires a description of the syntactic format of a message.

When the gateway connects two time-triggered networks, the a priori knowledge about the send and receive instants is available through port specifications. If the interconnected DASs use different periods or if they follow different paradigms, the gateway requires buffering functionality.

3.8.3 Encapsulation

Another purpose of a gateway is the encapsulation of a DAS. Not every message exchanged within a DAS needs to be visible outside the DAS. This helps under-

standing the complexity of a DAS and reduces bandwith costs between DASs. Encapsulation is also used to perform error containment by preventing the propagation of faulty messages through the gateway.

Selective Redirection

If the gateway has the ability to decide whether messages are forwarded or blocked, we talk about selective redirection. This requires filtering mechanisms in the temporal and value domain to be applied. In the value domain, the gateway checks the message contents (i. e. user data, message names and types). In the temporal domain, the temporal patterns of the traffic is monitored [OPK05a].

Complexity Control

Selective redirection reduces the messages that are visible outside a specific DAS. This feature also helps to minimize the mental effort required to understand a DAS and therefore reduces the complexity of the system.

Error Containment

Fault Containment Regions (FCRs) are used to restrict the impact of a fault to a specific region, but if faulty messages are exchanged between DASs, erroneous data can propagate across the boundaries of an FCR. In order to provide error containment to avoid the propagation of erroneous messages to another DAS, the selective redirection of messages at a gateway must be controlled by error detection mechanisms [OPK05a].

Chapter 4

A Model for an Integrated Architecture with Automated Tests

In this chapter, we develop a model for our validation framework but before we can do that, we need to take a look at the logical structure of DECOS which is our target architecture. Then we introduce the fault model to describe which types of faults we want to cover with our experiments. Finally, in section 4.5, we present the actual model of our framework.

4.1 Logical Structure of the DECOS Architecture

As already mentioned in section 3.2, the DECOS architecture provides a framework for the development of distributed embedded real-time systems. The overall system is divided into nearly-independent distributed application subsystems (DASs), possibly of different levels of criticality. Every DAS has its own dedicated computational and communication resources.

4.1.1 Services of an Integrated Architecture

There are a number of architectural services that separate the applications from the underlying platform technology. In order to maximize the number of platforms and applications that can be covered, those services are grouped into a minimal set of *core services* and into an open-ended number of *high-level services* that are built on top of the core services. This abstraction is depicted in figure 4.1

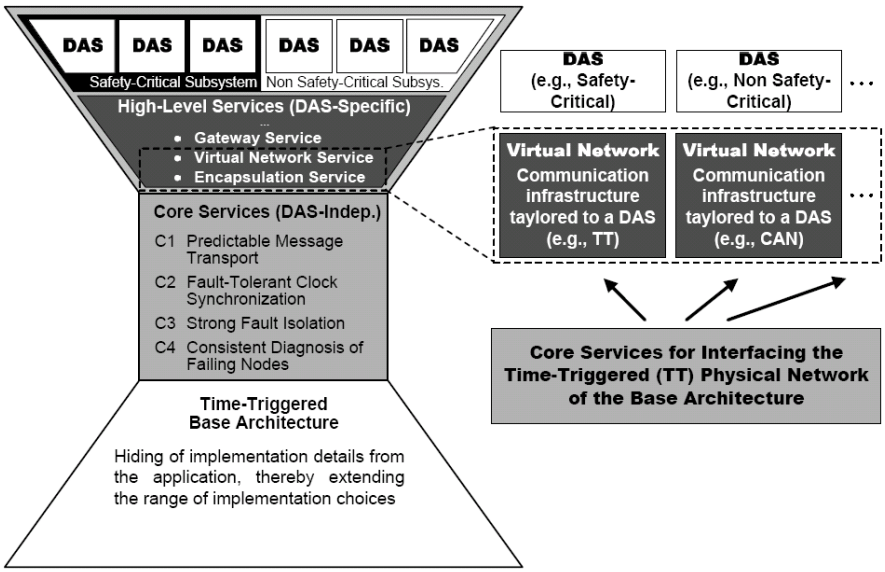


Figure 4.1: DECOS Integrated architecture [OPK05b]

Core Services

A platform for an architecture that supports the integration of multiple DASs with different levels of criticality must support the safety requirements of the highest considered criticality class. According to [OPT07], the core architecture for safety-critical real-time systems must provide the following services:

Deterministic and Timely Transport of Messages: This service is responsible for the transport of state messages from the sender's communication network interface (CNI) to the CNI of the receiver. The timely transport of messages with minimal latency and jitter is crucial for control stability in real-time applications [OPT07].

Fault-Tolerant Clock Synchronization: Because of the fact that the clocks of different nodes will drift apart with the progression of time, it is necessary to perform synchronization algorithms on the clocks to keep them in close relation with respect to each other. Clock synchronization is a fundamental service in a time-triggered system [OPT07].

Strong Fault Isolation: Fault Containment Regions (FCRs) are used to restrict the impact of a fault to a specific region, but fault effects manifested in erroneous data can propagate across FCR boundaries. To avoid error propagation, the system must also provide error

containment by applying error detection mechanisms. The error detection mechanisms must be part of a an FCR different from the sender of the erroneous message. Therefore, at least two FCRs are necessary to perform error containment. The set of FCRs that perform error containment is called Error Containment Region (ECR) [OPT07].

Consistent Diagnosis of Failing Nodes: A membership service at component level keeps track of the operational state of nodes in the cluster. In a time-triggered system, the receiver knows a priori when messages are about to arrive and therefore, the arrival of messages is interpreted as a life sign from the sending node.

Any architecture that provides these core services can be used as the core architecture of the DECOS integrated architecture. Examples of suitable architectures are the Time-Triggered Architecture (TTA) [Kop98] and FlexRay [Fle05].

High-Level Services

The high-level services are based on the core services. They are DAS specific and provide the interface for the jobs (cf. section 5.3.5) to the underlying platform.

Encapsulation Service: This service is responsible for spatial and temporal error containment [Rus99] at component level. We distinguish between error containmemt between the safety-critical and the non safety-critical subsystem and error containment between jobs at each subsystem.

Virtual Network Service: cf. section 3.7

Gateway Service: cf. section 3.8

Fault-Tolerance Service: In order to remain available despite the occurrence of component failures, an application service can be implemented by a group of redundant jobs at independent components. A group can mask failures of its members if the number and types of failures is covered by the failure mode assumptions. A prerequisite for systematic fault-tolerance is replica determinism [Pol96] by the architecture and the application [OPT07].

Diagnostic Service: This service is an extension of the core diagnostic service. We can distinguish between diagnostic acquisition service that monitors the port state of each job, and the diagnostic dissemination service that is forwarding information for a subsequent analysis

and must not depend on any control signals from the information consuming system.

4.1.2 Virtual Networks

Virtual networks provide the communication infrastructure for DASs. Each DAS has its own dedicated virtual network that fits its specific requirements. We can distinguish between time-triggered virtual networks and event-triggered virtual networks (3.7).

The interface between the virtual network and the applications running on it are called *links*, consisting of one or more *state* or *event ports*.

A *port* is dedicated to the transport or reception of a single message. A port specification captures the syntactic and temporal properties of the message instances and has a statically defined data direction (input or output port).

4.1.3 Jobs

Jobs are basic units of work that are scheduled periodically. They are host to the user application of a DAS that can be distributed across jobs in different nodes of the cluster.

A Job can access the communication infrastructure by fetching links that contain the required port specifications of the input or output ports the job is intended to send to or to receive messages from.

4.1.4 Gateways

In section 3.8 we defined gateways as a coupling between subsystems and distinguished three different kinds of gateways, namely, *hidden gateways*, *virtual gateways* and *physical gateways*.

In our model of an integrated architecture we use *hidden virtual gateways* in order to interconnect virtual networks at the architectural level, transparent to jobs at the application level.

Within a gateway, we can distinguish three different parts [OHP07]:

> The **Feeder Network Adaptor** receives messages from ports towards the virtual network where the sender of the message is located. It then dissects a message into its *convertible elements* (i. e. parts of a message that need no further subdivision) and pushes them into the gateway repository.

The **Gateway Repository** is a real-time database that stores the convertible elements of messages sent via the gateway. It is also responsible for resolving property mismatches in the value domain. The resolving of property mismatches in the time domain is within the responsibility of the network adaptors.

The **Retrieval Network Adaptor** is responsible for constructing the messages from convertible elements of the gateway repository. It then transmits the message via a port towards the receiving virtual network according to the properties from the port specification.

4.2 Communication Model

In our communication model, we differentiate between three types of messages, namely, periodic messages, sporadic messages and aperiodic messages.

Periodic messages are sent at equidistant points in time. The time between two message transmissions is called a period. This communication model is typical for time-triggered networks, where message transmissions are scheduled at a priori defined points in time.

Sporadic messages are typical for event-triggered networks where message transmission is triggered by events at unknown points in time.

When we consider messages without constraints regarding their timing, we talk about **aperiodic messages**.

4.2.1 Message Bursts

When a sender is allowed to send multiple messages in immediate succession, this is called a messages burst. The sender then has to maintain a specified inactivity interval before starting the next burst.

The activity interval where the sender is allowed to send a maximum number of *burstcount* messages is called *burst time*. We refer to the inactivity interval as *silence time*.

4.3 Fault Model

In the fault model, we describe which faults we take into consideration and the computer system should be able to tolerate. Therefore we need to specify fault

containment regions (FCRs) and error containment regions (ECRs).

Faults can be categorized into software faults and hardware faults.

4.3.1 Software Faults

Looking at the software of a node computer, we can distinguish between system software (operating system) and application software (jobs). All software on a node computer is dependent on the correct behavior of the system software and therefore, *all node computers on which a particular system software is deployed represent a common FCR for software faults affecting the system software* [OP06].

For software faults affecting the application software, we regard a software component as an FCR [OP06]. A software component component consists of one ore more jobs and my be replicated across several nodes. All replicas depend on the same program and inputs and can not be assumed to fail independently. Figure 4.2 shows the fault containment regions and the error containment region that is formed by a faulty job along with the gateway that prevents error propagation.

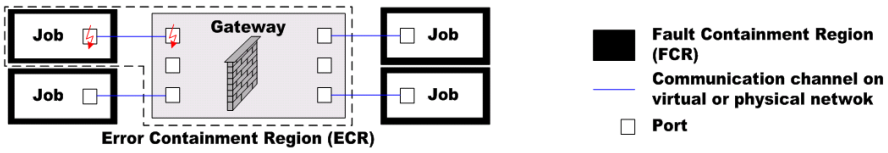


Figure 4.2: Fault and Error Containment Regions for Software Faults [OHP07]

We distinguish between two types of failures resulting from software faults:

> A **timing message failure** occurs when information is written into a port at an unspecified point in time.
>
> A **value message failure** occurs when the contents of a message that is written into the port do not comply with the specification. These incorrect message contents can encompass the message name (for explicit names only) and/or the message data (implicit and explicit names).

4.3.2 Hardware Faults

A hardware fault affects physical resources like mechanical or electrical components. They originate from development (design faults) or from conditions during operation (e. g. wearout or perturbations like electro-magnetic interference).

A node computer contains shared physical resources (e. g. processor, memory, power supply oscillator) and if any of this resources is corrupted by a fault, several or all of the software components running on a node could be affected. Therefore, we regard every node computer along with the software components running on that node as a fault containment region (cf. figure 4.3) [OP06]. The failure modes are the same as they are for software faults, only, they affect multiple ports.

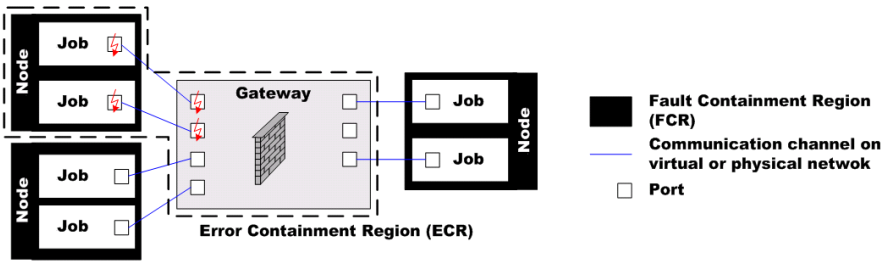


Figure 4.3: Fault and Error Containment Regions for Hardware Faults [OHP07]

4.4 Error Containment

In section 4.1.4, we distinguished three different parts of a gateway, the feeder network adaptor, the retrieval network adaptor and the gateway repository, each of which can exert error containment mechanisms (cf. figure 4.4).

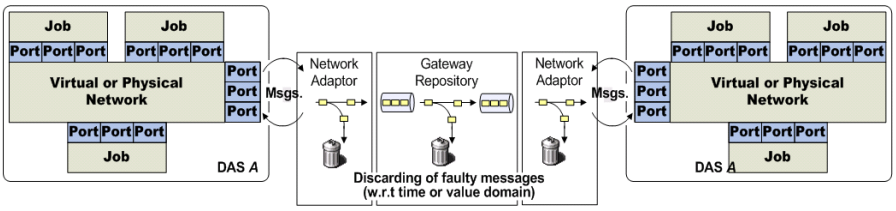


Figure 4.4: Error Containment Stages in the Gateway [OHP07]

4.4.1 Spatial Partitioning

Spatial partitioning (cf. section 3.6) is concerned with error containment with respect to value message failures. In the fault model 4.3, we defined a *value message failure* as the case when a the content of a message written into a port does not comply with the specification. If a message contains an explicit message name, a *timing message failure* could lead to spatial interference:

An **invalid source address** results in a masquerading failure because the origin of the message is falsified.

An **invalid destination address** can be the cause of harmful effects at the unintended receiver while the intended receiver never receives the message.

If the **message identifier** is corrupted, the content of a message is likely to be misinterpreted (e. g. a temperature value mistaken for a speed value).

Spatial Partitioning using Message Objects

A gateway receives messages from a virtual network and dissects its contents into convertible elements. The network adaptors of the gateway are implemented as *timed gateway automata* (TGA) and the convertible elements are stored as TGA variables. The messages that are sent by the gateway are assembled from a set of those TGA variables. We define a message object by the following tuple:

$$\text{Message object: } < \text{Ports, TGA Variable, Name} >$$

The element *ports* specifies the ports, where the messages are sent and received. *TGA variable* defines the content of the message and *name* is used to distinguish message objects. The gateway is provided with information about the structure of the messages is handles at startup. It uses this information to compare the structure messages with the intended structure during operation and discards messages that do not conform with the specification. Figure shows an example of this process for a set of CAN messages. Two TGA variables accept messages from the same port, but exhibit different message names. A third message which is located in the port is discarded, since it is not associated with a TGA variable.

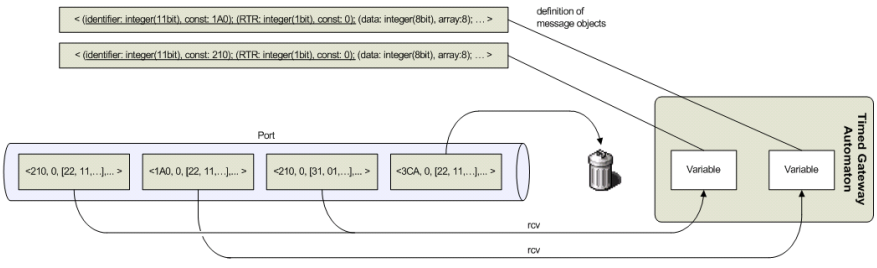


Figure 4.5: Example for Spatial Partitioning by Feeder Network Adaptor [OHP07]

Spatial Partitioning using TGA

The use of TGA allows to specify criteria whether messages should be forwarded or discarded by the gateway. Figure 4.6 depicts the simplest case of spatial partitioning realized using a TGA in the feeder network adaptor. The TGA specifies a transaction that starts with a message reception to acquire a message from a port (i. e. , transition label "$rcv(m)$") and ends with a push operation to forward the information into the gateway repository (i. e. , transition label "$push(m)$"). In between, the transaction executes a check (i. e. , transition label "$f(m)$") on the message contents that determines whether the message is discarded.

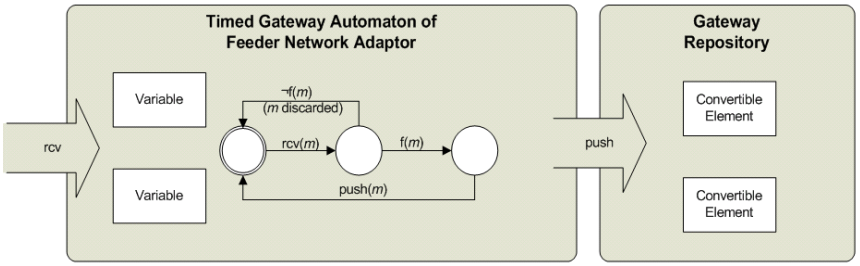


Figure 4.6: Partitioning in Feeder Network Adaptor using TGA [OHP07]

Spatial partitioning at the retrieval network adaptor works in the same manner, only that the specified criteria are applied to convertible elements pulled from the gateway repository. A simple automaton is shown in figure 4.7.

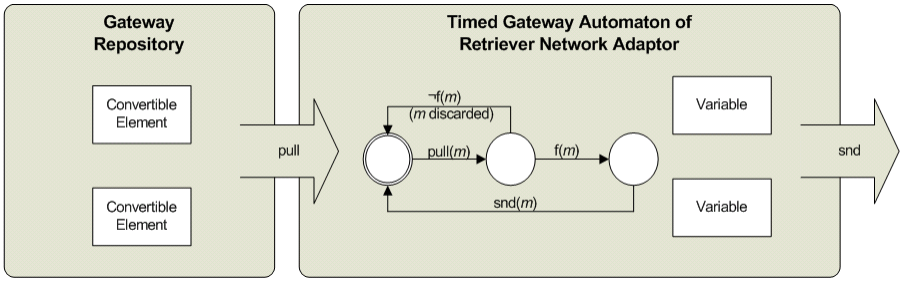


Figure 4.7: Partitioning in Retrieval Network Adaptor using TGA [OHP07]

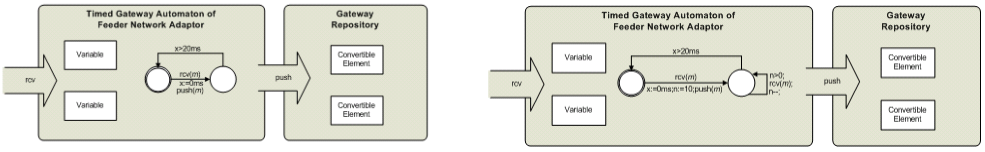
The gateway repository has the ability to perform syntax conversion by the use of a so called *transfer syntax*. If the arrival of a new convertible element triggers a conversion, the target convertible element is computed. Criteria can be specified whether an update of the target element should be performed or not.

4.4.2 Temporal Partitioning

Error containment with respect to *timing message failures* results in temporal partitioning (cf. 3.6). The gateways preserve predefined temporal properties (i. e. bandwidths, latencies) of the interconnected networks despite the redirection of messages.

Temporal Partitioning at the Feeder Network Adaptor

The feeder network adaptor has a priori knowledge about minimum interarrival time and discards messages that represent timing message failures (a message that violates interarrival times). Figure 4.8 shows two examples for temporal partitioning using TGA in the feeder network adaptor. The TGA on the left hand side (cf. 4.8(a)) maintains an interarrival time of 20 ms, meaning that a receive operation with a subsequent push into the repository starts a silence interval of 20 ms. The TGA on the right hand side (cf. 4.8(b)) supports 10 message receptions in immediate succession and enforces an inactivity interval of 20 ms between any two bursts.



(a) minimum interarrival time (b) minimum interarrival time with burst

Figure 4.8: Temporal Partitioning in Feeder Network Adaptor using TGA

Temporal Partitioning at the Retrieval Network Adaptor

The retrieval NWA reads from the gateway repository and uses a priori knowledge about temporal properties to discard messages representing timing message failures. The TGA of the retrieval NWA maintains interarrival times and supports burst like the TGA of the feeder NWA.

Temporal Partitioning at the Gateway Repository

The gateway repository supports separate buffering of different convertible elements and a timing message failure resulting in the overload of one convertible element does not affect the other convertible elements.

4.5 A Framework for Automated Tests

As we already stated in the introduction, our approach for convenient experimental evaluation is the development of a framework that is capable of automatic execution of tests and the ability to monitor the system under test. In our model, the framework consists of a test controller and monitor and an execution environment in which the tests are executed. The model is depicted in figure 4.9.

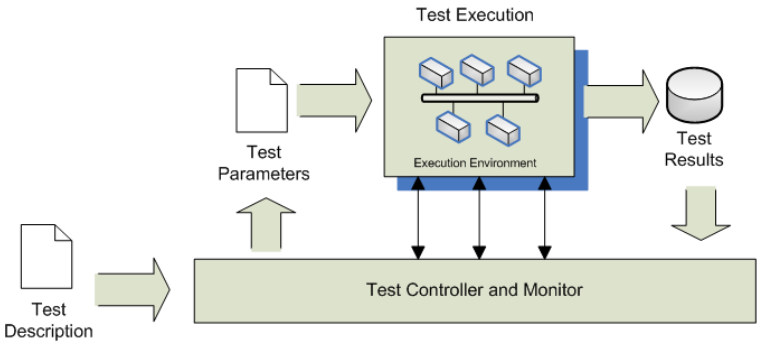


Figure 4.9: Model of a Test

Test cases are manifested in a test description that serves as input to the test controller. The controller then feeds the execution environment with the parameters from the test description and controls and monitors the target hard- and software at runtime. At the end of a testrun the controller fetches the gathered measurements and analyzes the results.

The following paragraphs describe how fault injection and monitoring are performed at runtime.

4.5.1 Fault Injection

We want to evaluate the behavior of the system in the presence of faults and therefore, an important requirement for a validation framework is the ability to induce faults into the system. We pursue the approach of software implemented fault injection at runtime [Ade03].

SWIFI at Runtime via Jobs

The target application jobs send and receive messages according to the parameters they get from the test controller at startup. Faults are injected into the system by providing faulty parameters to the jobs. The following parameters can be specified:

Initial Delay specifies the point in time when a particular message is sent for the first time.

Fixed Delay is a constant constant period of time between the transmission of two consecutive messages.

Random Delay allows us to randomize the gap between message transmissions. *Fixed delay* and *random delay* together specify periodic, aperiodic or sporadic behavior.

Direction specifies the data direction for a particular message.

Mode specifies if a message should be sent only once (one-shot mode) or repeatedly (continuous mode).

Message ID identifies a message.

Message Length indicates the length of the message in bytes.

The parametrization of the message transmission behavior of jobs allows us to create a wide variety of fault scenarios without changing the target application's source code. By setting the parameter *initial delay*, we can specify the point in time when a fault is injected into the system. By setting *Fixed delay* and *random delay* accordingly, we can achieve periodic, aperiodic or sporadic behavior and allows us to produce timing message failures. Naming message failures are produced by setting wrong *message ids* or *message lengths*.

4.5.2 Monitoring

Our framework provides the possibility to monitor the system under test. The monitoring service is integrated within the target application. The application jobs produce records for every send and receive operation that are analyzed by the test controller after the test has finished. The jobs also have the possibility to send diagnostic messages to the controller that are sent over a separate communication channel instead of the tested network. This situation is illustrated in figure 4.10.

4.5.3 Requirements

In the introduction, we identified three major requirements our framework has to meet, namely:

- Monitoring of the system under test without probe effects

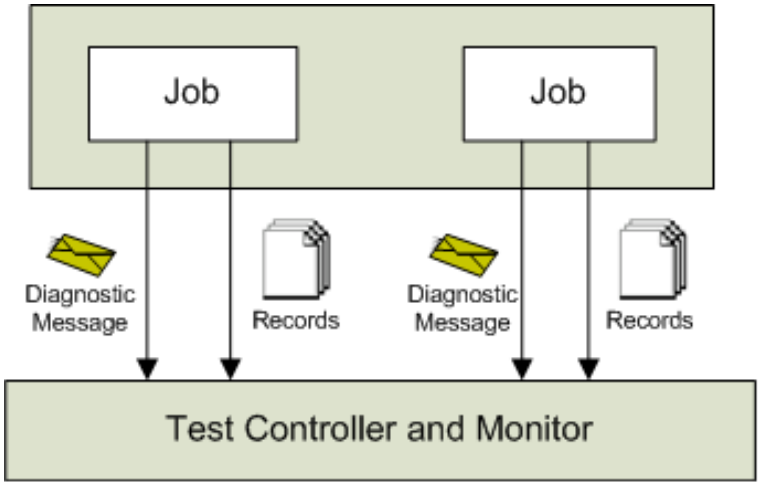


Figure 4.10: Monitoring Access

- The ability to induce faults into the system
- Reproducibility of tests

As mentioned in section 4.5.2, monitoring is performed by the application jobs running on the execution environment. The measurements of an experiment are fetched by the test controller after the experiment has finished execution and not transmitted during the test in order to prevent probe effects.

The message formats, contents and timings are derived by the application jobs from parameter files provided by the test controller. Fault injection is achieved by providing faulty parameters, representing naming or timing message failures, to the jobs.

The parameters needed for the execution of a specific testcase are derived from files containing testcase descriptions. Those testcase descriptions are stored on a server which allows us to reproduce tests anytime and by anyone.

Chapter 5

Implementation of the Framework

This chapter begins with a description of the prototype implementation of a DECOS cluster. Section 5.1 describes the hardware of the DECOS cluster, while section 5.2 is dedicated to the node software, including the operating system, virtual networks and gateways, as well as the execution environment for the application software. The rest of this chapter is dedicated to the measurement framework that has been set up to evaluate the virtual gateways of the prototype implementation.

5.1 Cluster Hardware

This implementation comprises a cluster with five integrated components, interconnected by a physical TTP core network. Each of those components consists of three single board computers, using 100 Mbps Ethernet for component-internal communication. One of them implements the Basic Connector Unit (BCU) and provides the DAS independent core services of such an integrated architecture. The other two represent the secondary connector units, one for the safety-critical subsystem (SCU) and one for the non safety-critical subsystem (XCU). They host the application soft-ware and provide DAS specific high-level services. The cluster is depicted in figure 5.1.

The basic connector unit hardware is a TTTech monitoring node [TTT02b], equipped with an embedded PowerPC processor from Motorola with on-chip fast Ethernet for component internal communication and a TTP/C [TTT02a] communication controller for time-triggered communication with other components.

For the secondary connector units we use the Soekris Engineering net4521 single board computers. They are based on a 133 MHz 486 class Elan processor from AMD. They have two 10/100 Mbit Ethernet ports, 64 Mbyte SDRAM main memory and use CompactFlash modules for program and data storage. Two

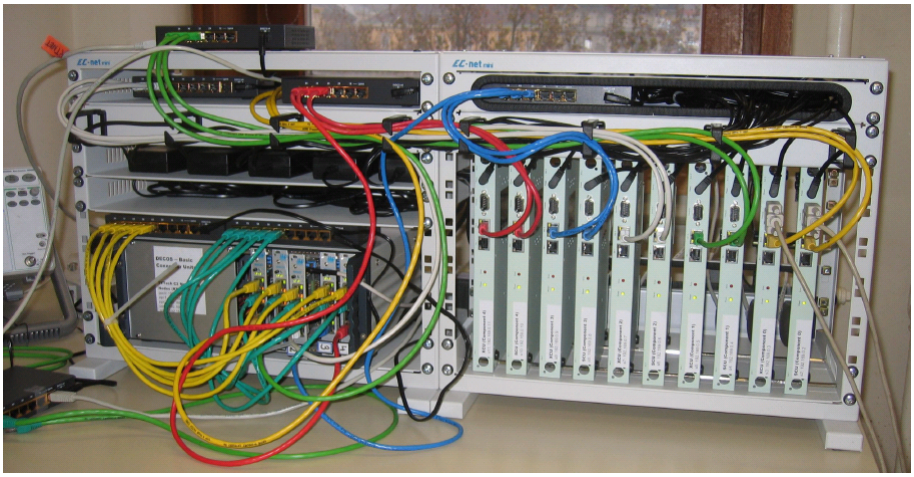


Figure 5.1: DECOS Prototype Cluster [HPOS05]

PC-Card/Cardbus adapters allow us to equip the nodes with a PCMCIA-based LIN/CAN interface card that can thus work as physical gateways to a physical CAN or LIN network.

Every target computer has an NFS connection to a server from where it can fetch the latest applications. This connection can also be used for monitoring purposes. The monitoring access is described in detail in section 5.3. Figure 5.2 shows the interaction between BCU, SCU and XCU in a DECOS component and the connection to the server.

5.2 Node Software

The node software consists of the operating system, the real-time scheduler, the high-level services (VN Service, Gateway Service, Task Wrapper) and the user applications. Every part will be described in detail in the following subsections.

5.2.1 Operating System

Every node, BCUs as well as secondary connector units, uses the embedded real-time variant RTAI (Real Time Application Interface) [BBD+00] as its operating system. It combines a real-time hardware abstraction layer (RTHAL) with a real-time application interface, thus making Linux suitable for hard real-time applications. In our case, RTAI v3.1 on a Linux 2.6 Kernel including the real-time RTnet Ethernet driver suite is used. While in the BCUs only kernel modules are used, the net4521 nodes make use of the partitioning capabilities of the RTAI/LXRT

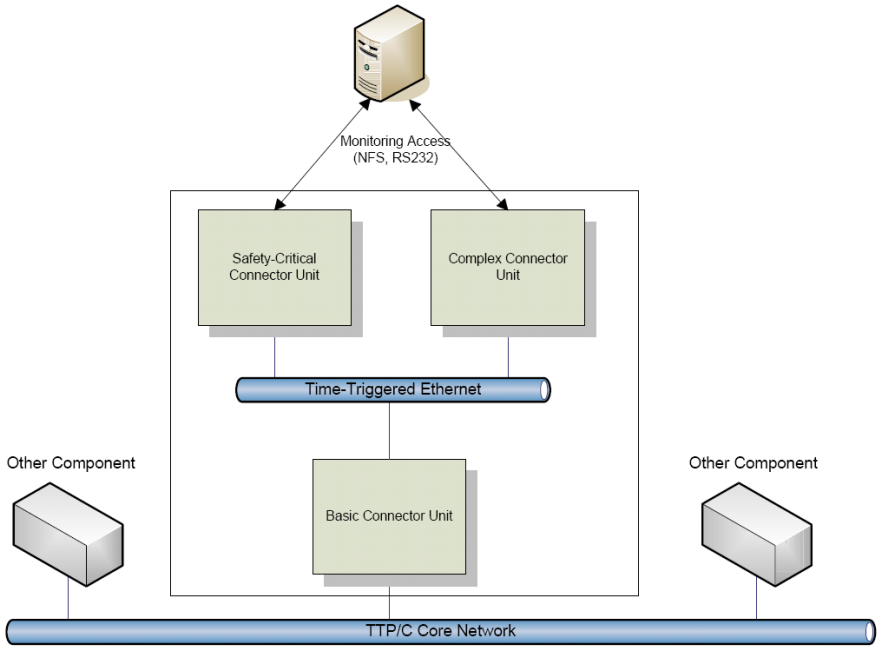


Figure 5.2: A DECOS Component

extension. LXRT enables the development of hard real-time programs running in user space instead of kernel space. Furthermore, LXRT eases the communication between hard real-time and non real-time processes, which can be utilized for monitoring and debugging of the hard real-time processes, without affection their real-time behavior [HPOS05]. In case of software failures within a job, the LXRT-based execution environment prevents the software in one partition from disrupting the software of any other partition, i. e. partitioning inhibits the propagation of software failures between jobs. According to the fault hypothesis of the DECOS integrated architecture, a job is regarded as FCR for software faults [OPT07]. In our prototype implementation, secondary connector units and application computer share the same computational resources. Thus, in case of a software failure of one or more jobs, partitioning ensures the undisturbed execution of the high-level services as well as of the remaining jobs.

5.2.2 Virtual Networks

The virtual network configuration [OPK05b] that has been set up on the prototype cluster for the evaluation process consists of four virtual networks. There are two safety-critical time-triggered networks and two event-triggered networks. The safety-critical TT networks are called By-Wire DAS and Navigation DAS. The

Network	Description	Paradigm	Criticality
NETWORK_BYWIRE	By-Wire DAS	TT	safety-critical
NETWORK_COMFORT	Comfort DAS	ET	non safety-critical
NETWORK_NAVIGATION	Navigation DAS	TT	safety-critical
NETWORK_LIGHTS	Lights DAS	ET	non safety-critical

Table 5.1: Virtual Network Configuration

ET networks are both non safety-critical and are called Comfort DAS and Lights DAS. The messages exchanged over the ET networks are either CAN messages (Extended CAN Message format) [Bos99] or IP messages. Table 5.1 gives an overview of the virtual networks named above. They are all intended to apply to typical applications in the automotive industry.

5.2.3 Global Time Service

The DECOS integrated architecture uses a uniform 64 bit long time format which is closely related to the GPS time format. It has been standardized by the OMG in the smart transducer interface standard [OMG, 2003]. The smart transducer time format has a granularity of 2-24 seconds (about 60 nanoseconds) and a horizon of 240 seconds (more than 30000 years). The epoch starts with the epoch of the GPS time i. e. , January 1980. The global time is accessible by the user applications through a shared memory on every component.

5.2.4 Gateways

In the exemplary setup, only hidden virtual gateways [OPK05a] are used. The gateways are listed in table 5.2. Table 5.3 shows the communication partners for the gateways.

The following paragraphs will describe the gateways in detail.

Gateways between two time-triggered networks

This type of gateway interconnects two time-triggered networks (By-Wire DAS and Navigation DAS), so no queues are necessary. The Feeder NWA reads the state variable from the input port and updates the convertible element (a 4 byte sequence number) in the repository. The Retrieval NWA reads out the convertible element from the repository and updates the state variable in the output port.

Gateway	Feeder Network	Retrieval Network	Component	Job	Links and Ports
TT1_to_TT2	By-Wire DAS	Navigation DAS	SCU3	job1_bywire_gw	Link1_VN3_Bywire: 0 Link1_VN2_Navigation: 0
ET1_to_ET2	Comfort DAS	Lights DAS	XCU2	job1_comfort_gw	Link1_VN0_Comfort: 0 Link1_VN1_Lights: 0
TT1_to_ET1	By-Wire DAS	Comfort DAS	XCU4	job5_bywire_gw	Link5_VN3_Bywire: 0 Link5_VN0_Comfort: 0
ET1_to_TT1	Comfort DAS	By-Wire DAS	SCU0	job7_comfort_gw	Link7_VN0_Comfort: 0 Link7_VN3_Bywire: 0
RR_ET_to_TT	Comfort DAS	Comfort DAS	XCU2	job8_comfort_gw	Link8_VN0_Comfort: 0 Link8_VN0_Comfort: 1 Link8_VN3_Bywire: 0
IP_GW	Comfort DAS	Lights DAS	XCU3	job9_comfort_gw	Link9_VN0_Comfort: 0 Link9_VN1_Lights: 0
IP2CAN	Comfort DAS	Lights DAS	XCU3	job9_comfort_gw	Link9_VN0_Comfort: 1 Link9_VN1_Lights: 1
CAN2IP	Comfort DAS	Lights DAS	XCU3	job9_comfort_gw	Link9_VN0_Comfort: 2 Link9_VN1_Lights: 2

Table 5.2: Gateway Setup

Sender	Component	Message / Size	Gateway	Receiver	Component	Message / Size
job0_bywire	SCU0	msg0_bywire / 4	TT1_to_TT2	job0_navigation	SCU2	msg0_navigation / 4
job0_comfort	XCU0	msg0_comfort / 12	ET1_to_ET2	job0_lights	XCU3	msg0_lights / 12
job4_bywire	SCU3	msg2_bywire / 4	TT1_to_ET1	job4_comfort	XCU3	msg2_comfort / 12
job6_comfort	XCU1	msg3_comfort / 12	ET1_to_TT1	job6_bywire	SCU2	msg3_bywire / 4
job0_comfort	XCU0	msg0_comfort / 12	RR_ET_to_TT	job0_comfort	XCU0	msg4_comfort / 12
job0_comfort	XCU0	msg5_comfort / 28	IP_GW	job8_lights	XCU3	msg5_lights / 28
job0_comfort	XCU0	msg5_comfort / 28	IP2CAN	job8_lights	XCU3	msg6_lights / 12
job0_comfort	XCU0	msg0_comfort / 12	CAN2IP	job8_lights	XCU3	msg5_lights / 28

Table 5.3: Communication Partners at Gateways

Gateways between two event-triggered networks

There are four gateways that interconnect the two ET networks (Comfort DAS and Lights DAS), one for CAN messages (named ET1_to_ET2) and the other one for IP messages (IP_GW, IP2CAN and CAN2IP). At ET1_to_ET2 a semantic property mismatch occurs, because the CAN message from the feeder network has another identifier than the CAN message that has to be sent on the retriever network (because of different name spaces). This mismatch is resolved by storing the appropriate identifiers at the corresponding NWAs.

The Feeder NWA consumes messages from the queue at the input port, extracts the convertible element (a 4 byte sequence number) and pushes it into the repository (with queue length 10). The Retrieval NWA pulls a convertible element from the repository creates an ET message with the appropriate syntactic (CAN or IP) and semantic (correct CAN identifier) properties and inserts it into the queue at the output port.

Gateways connecting time-triggered and event-triggered networks

Event-triggered to time-triggered

Two different control paradigms have to be combined at this gateway. As in the ET to ET gateways, the Feeder NWA receives messages from an incoming queue and pushes the convertible element into the repository (queue length 10). When a convertible element is available in the repository, the Retrieval NWA updates the state variable in the output port. The difference is, that, according to the TT paradigm, a message containing the state variable is sent across the network in every TDMA round anyway.

Time-triggered to event-triggered

This gateway combines a TT and an ET network (By-Wire DAS and Comfort DAS). The Feeder NWA receives a state message in every activation round and updates the data in the repository in place (no queue). The Retrieval NWA has to recognize a change in the state variable as an event, create an ET message and inserts it into the outgoing queue.

Gateways implementing a client/server architecture

This type of gateway implements a client/server architecture where the gateway acts as the server. It receives requests from the ET network (Comfort DAS) and answers with a response message containing the required data (the state variable from the TT network (By-Wire DAS)). In this case we can't really talk of a Feeder and a Retrieval NWA. Instead, there is one network adaptor that receives state

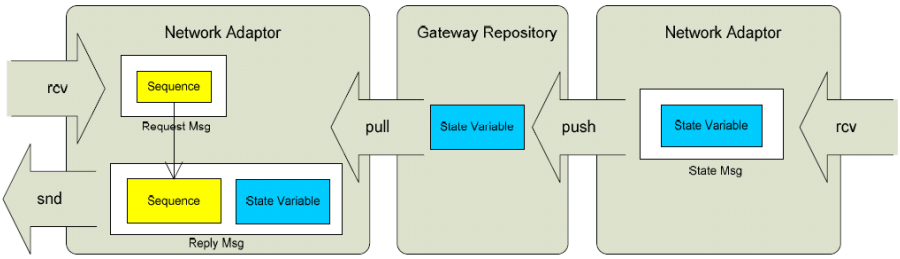


Figure 5.3: Functionality of the Request/Reply Gateway

messages from the TT network and updates the repository once per TDMA round, while the other network adaptor implements the request/reply mechanism. Every request message contains a sequence number that has to be packed into the reply message along with the real-time data. The behavior of the gateway is depicted in figure 5.3.

The spatial partitioning mechanisms are implemented at the Feeder NWA. Therefore two messages are stored at the gateway, a reference message and a mask message. The message received from the input port is masked with the mask message and then compared to the reference message. If there is a mismatch in the length or in the message name, the message will be discarded. If the message passes the comparison, it will be processed by the gateway.

In order to perform temporal partitioning, the Feeder NWA makes use of the global time. The gateways were created with the Gateway Code Generator (gwcg) [H06] and were enhanced with a so called burst mode. This allows the gateway to limit the bandwidth across virtual networks and to enforce different timing schemes of alternating burst and silence periods. Figure 5.4 shows the automaton of the Feeder NWA. It can be seen that the gateway starts in the waiting state. If a message is available in the incoming queue, a new burst starts and *burstTime* and *silenceTime* are set using the predefined values *BURST_TIME* and *SILENCE_TIME* from a global header file. A burst lasts until *burstTime* has elapsed and the GW accepts *BURST_COUNT* at maximum. Any messages exceeding that value are deleted (i. e. removed from the queue without forwarding). When in *silence* the gateway does not accept any messages. An overview of the important parameters can be found in table 5.4. Since the global time is updated only once per activation, the burst and silence times are limited to multiples of the TDMA round length (10 ms) of the cluster.

The source code of the Feeder NWA automaton of an ET to TT gateway can be found in listing 5.1. At the beginning of every activation, *currentTime* is updated with the global time. The initial state is waiting (line 8), where *burstTime* and *silenceTime* are set. State *burst_action* (line 20) is where the receive operation takes place and the convertible element is pushed into the repository, if the reception

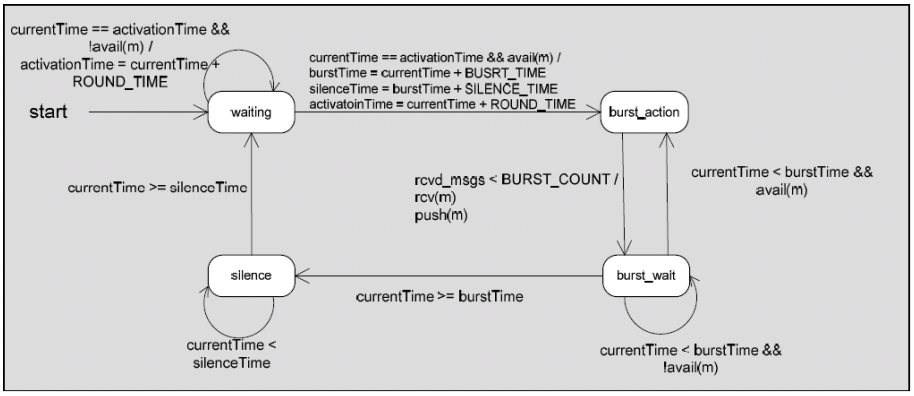


Figure 5.4: Timed Gateway Automaton of the Feeder Network Adaptor

Parameter	Description
BURST_TIME	duration of burst in μs
SILENCE_TIME	duration of silence in μs
BURST_TIME	maximum number of messages per burst
ROUND_TIME	duration of one TDMA round

Table 5.4: Parameters for Burst Mode

succeeded and the state is switched to *burst_wait* (line 38). As long as the burst is active and messages are available in the incoming queue, the gateway toggles between *burst_action* and *burst_wait*.

```
1  static void atm0_number_export(void) {
2
3    short guardEnabled = 0;
4    currentTime = shmTime->time;
5
6    do {
7      switch (state) {
8      case waiting:
9        if (avail_m(&(gw_messages[0]))) {
10         avail_total++;
11         burstTime = actTime + US_TO_GT64(BURST_TIME);
12         silenceTime = burstTime + US_TO_GT64(SILENCE_TIME);
13         guardEnabled = 1;
14         rcv_msgs = 0;
15         state = burst_action;
16       } else {
17           guardEnabled = 0;
```

```
18        }
19        break;
20      case burst_action:
21        if (rcv_msgs < BURST_COUNT) {
22          if (rcv(&(gw_messages[0]))) {
23            atm0_rep_ce000=atm0_m000_ce000;
24            push(&(atm0_rep_ce000));
25            rcv_msgs++;
26          }
27        } else {
28            // remove msg
29            while(avail_m(&(gw_messages[0]))) {
30              avail_total++;
31              rcv(&(gw_messages[0]));
32              rcv_total++;
33              remove_total++;
34            }
35        }
36        state = burst_wait;
37        break;
38      case burst_wait:
39        if (currentTime < burstTime) {
40            if (avail_m(&(gw_messages[0]))) {
41              avail_total++;
42              state = burst_action;
43              guardEnabled = 1;
44            } else {
45              guardEnabled = 0;
46            }
47        } else {
48            state = silence;
49            guardEnabled = 1;
50        }
51        break;
52      case silence:
53        if (currentTime >= silenceTime) {
54          guardEnabled = 1;
55          state = waiting;
56        } else {
57          guardEnabled = 0;
58          // remove messages
59          while(avail_m(&(gw_messages[0]))) {
```

```
60            avail_total++;
61            rcv(&(gw_messages[0]));
62            rcv_total++;
63            remove_total++;
64            }
65          }
66          break;
67        default:
68          break;
69        }
70    } while (guardEnabled);
71  }
```

Listing 5.1: Gateway Automata

5.2.5 Execution Environment

The execution environment consists of a static scheduler and the task-wrapper that periodically executes high-level services and application jobs.

The scheduler is an RTAI kernel module. It runs in kernel space with highest system priority and is responsible for all LXRT tasks of its scope. If some task fails to finish within its associated time slot, the scheduler suspends this task to ensure that temporal partitioning can be guaranteed. For more details about the temporal partitioning in the DECOS prototype implementation see [HPOS05].

Every component in the cluster hosts three application jobs (job1, job2 and job3). The jobs run in user space and allow the user to access the virtual networks by fetching links according to the VN configuration.

Application jobs have two entry points, one called once for initialization named *init_point()* and another one called periodically named *entry_point()*. Those entry points are called by the task-wrapper that is linked with the jobs. What the application jobs actually do is part of the measurement application that is discussed in the following section 5.3.

More details about the execution environment of the prototype cluster and the high-level services can be found in [HPOS05] and [HÖ06].

5.3 The Measurement Framework

A measurement framework has been implemented in order to evaluate the virtual gateways of the prototype implementation described in the previous sections of this chapter. The results of the experimental evaluation are discussed in chapter 6. The framework consists of the following parts:

- A **control application** with a graphic user interface (GUI) that includes an XML parser and the analysis tool that is used to analyze the results of a test. It is also responsible for the coordination of a test.
- The **XML parser** of the control application takes XML files containing the specification of a test case as input and generates a file in the character separated value (CSV) format that is readable by the test run generator running on the target nodes.
- A **test run generator** is installed on the target nodes. It reads the parameters from a CSV file and writes them into a shared memory for every job running on the node.
- The **application jobs** act as senders and receivers of the system and act according to the parameters stored in the shared memory. They are also responsible for recording their operations for the analysis after the test.
- The control application comes with an **analysis tool** that gathers the records of all the test runs of a test and analyzes the properties we are interested in (i. e. minimum, maximum and average latencies, bandwidths, message loss).
- Test cases for the experimental evaluation are specified in **XML files** that provide the input for the control application
- **Monitoring access** to the target nodes is provided via the serial interface.

Figure 5.5 shows the data flow in the measurement framework, beginning with the XML specifications as input and resulting in the output of CSV files containing the statistics of a test.

The dataflow in a target node (SCU or XCU)during a test run is shown in figure 5.6. At cluster start-up, the file containing the parameters is fetched from the DECOS server via NFS. It is then read by the test run generator that writes the parameters into a shared memory. Every application job has its own dedicated

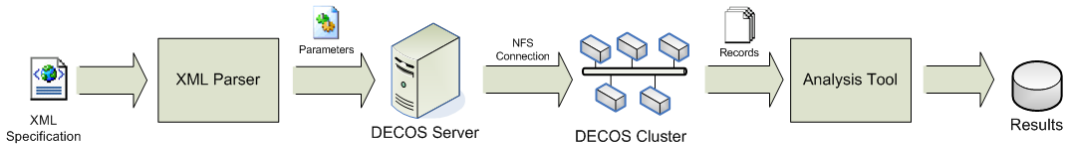


Figure 5.5: Data Flow in the Framework

shared memory. During a test run, the jobs send and receive messages according to the specified parameters and record their operations. At the end of a test run, the jobs write their records to a file that is copied to the DECOS server.

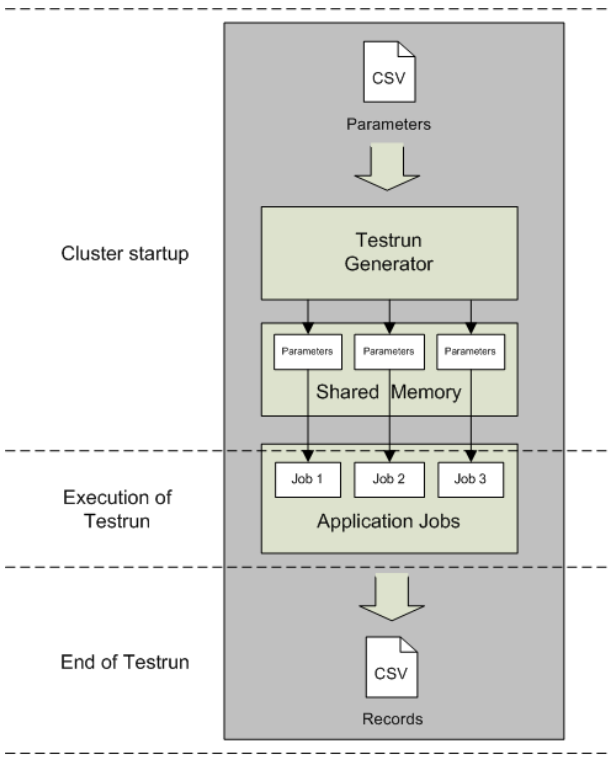


Figure 5.6: Data Flow at SCUs / XCUs

The following subsections describe the particular parts of the measurement framework in detail.

5.3.1 Control Application (GUI)

The control application for the measurement framework can be installed on any computer with a connection to the internet. It is written in Java and provides a graphic user interface (GUI) through which the user can specify, coordinate and

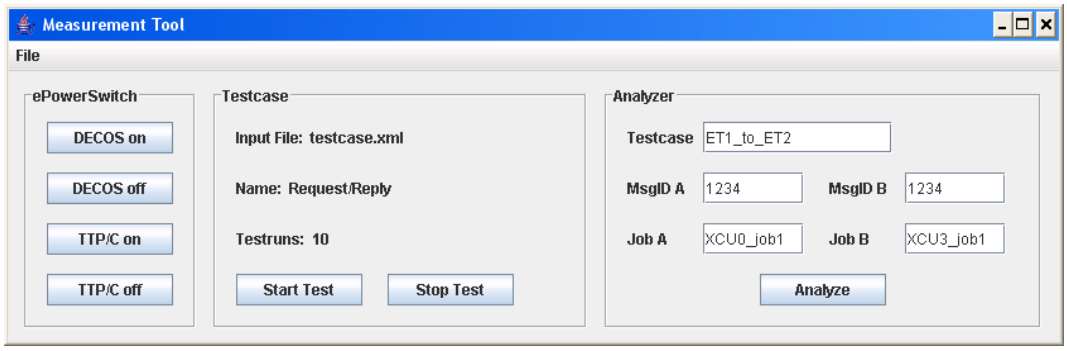


Figure 5.7: GUI of the Measurement Framework

monitor a test. A screenshot of the GUI is depicted in figure 5.7.

The user begins by opening an XML file containing the specification of a particular test case which is then processed by the application. The parameters are then stored in a local CSV file. If the user decides to start a test, the application opens an SSH (secure shell) connection to the DECOS server (decos.vmars.tuwien.ac.at) and uses the secure copy protocol (SCP) to transfer the parameters to the server. The cluster is switched on via a remote controlled power switch via IP or the RS232 interface.

At the end of a test run, the records are available at the DECOS server. The control application opens an SSH connection and creates a zipped file that contains the records of all nodes in the cluster. The file is copied to the local host via SCP and is then processed by the analysis tool. The analysis tool is described in section 5.3.6.

The control application can also be used to manually turn the DECOS cluster on and off via the remote controlled power switch.

5.3.2 XML Specifications

An XML specification contain all the parameters that are necessary to execute a test on the DECOS cluster. First of all, it contains the name of the test case and the number of test runs that specifies how often the test should be repeated. It may as well contain a brief description to describe the purpose of the particular test case for a human reader.

Every test run consists of at least one *message* and at least one *record*. A *message* element requires a *network* and at least one *link* in this network to specify the sending communication partner(s) for this type of message. The attributes of a *message* describe the syntactic format and the temporal properties for the message.

A *record* defines a communication partner that is intended to record receive operations. Like a *message*, it consists of a *network* and *links* in the network but has no further attributes.

A *link* has a *link_id* and consists of one ore more *ports* along with their *port_id*. If no ports are specified, every port of the link is used.

The networks, links and ports used in the specifications have to match the setup in the *virtual network configuration* of the cluster.

Listing 5.2 shows the document type definition (DTD) for the specifications.

```
<?xml version='1.0' encoding='utf-8'?>
<!ELEMENT testcase ((description)?, (testrun))>
<!ATTLIST testcase
    name        CDATA #REQUIRED
    nTestruns   CDATA #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!ELEMENT testrun ((message)+, (record)+)>

<!ELEMENT message ((link)+, (network))>
<!ELEMENT record  ((link)+, (network))>
<!ATTLIST message
    type            (REPEAT | ONE_SHOT) #REQUIRED
    initial_delay   CDATA #REQUIRED
    fixed_delay     CDATA #REQUIRED
    random_delay    CDATA #REQUIRED
    msg_len         CDATA #REQUIRED
    msg_id          CDATA #REQUIRED>

<!ELEMENT link ((port)*)>
<!-- if no port specified, select all ports of that link -->
<!ELEMENT port EMPTY>
<!ELEMENT network EMPTY>
<!ATTLIST link link_id CDATA #REQUIRED>
<!ATTLIST port port_id CDATA #REQUIRED>
<!ATTLIST network network_id CDATA #REQUIRED>
```

Listing 5.2: Document Type Definition of a Test Case

5.3.3 XML Parser

The control application includes an XML parser that is invoked whenever the user opens a test case specification. It is a SAX (Simple API for XML) parser that is capable of validating the input file against the document type definition (DTD).

After data extraction, the application stores the parameters in a CSV file that is transferred to the server at the begin of a test run and can be processed by the test run generator running on the target computers.

5.3.4 Test Run Generator

During the start-up phase of the DECOS cluster, the target nodes establish an NFS (Network File System) to the DECOS server in order to fetch the user applications as well as the parameters for the current test run.

The parameters for the test jobs for all nodes are contained in a single CSV file. Before the jobs are started, the test run generator is executed to process the file, extract the parameters, arranges them into *execution blocks* (cf. section 5.3.5) and writes them into a shared memory. Every job has its own dedicated shared memory location.

5.3.5 Application Jobs

The application jobs are the most important part of the measurement framework. They are time aware, by making use of the global time service discussed in section 5.2.3. The jobs are responsible for sending and receiving messages across the network and to log the send and receive operations.

The operations of a job are divided into so called *execution blocks*. Execution blocks are data structures located in shared memory location and contain parameters used by the job to derive the message format and the send instants used in the current test. The shared memories are written by the test run generator during start-up, read by the application jobs when the job's *init_point()* is invoked and then stored locally. Listing 5.3 contains the type definition of the execution blocks.

```
1 typedef struct exec_struct {
2
3   // temporal parameters
4   unsigned int initial_dly;
5   unsigned int fixed_dly;
6   unsigned int random_dly;
7
```

```
8     // spatial parameters
9     unsigned char msg_len;
10    unsigned int  msg_id;
11    unsigned char port_id;
12
13    // control part
14    unsigned char dir; // 0...in, 1...out
15    unsigned char mode;
16    unsigned char status;
17    gt64 send_instant;
18
19  } t_exec_block;
```

Listing 5.3: Type definition of Execution Blocks

The temporal parameters are *initial delay*, *fixed delay* and *random delay*. *Initial delay* tells the job, when the message has to be sent for the first time. *Fixed delay* and *random delay* allow us to achieve either periodic, aperiodic or sporadic send behavior. The spatial parameters for the messages are *message length*, *message ID* and *port ID*. In the control part of the execution block we can set the *direction* (in or out) and the *mode* (i.e. continuous ore one-shot). *Status* tells the job, if the execution block is still active. The data structure furthermore contains the next *send instant* for the message and its value is updated after every send operation based on the temporal parameters mentioned above. The missing parameters, *virtual network* and *link*, are passed to the job as arguments at start-up because they are needed to calculate the ID of the shared memory and can thus not be contained in the shared memory itself.

Execution of Test Run

Every time a job's *entry_point* is invoked, the send instant of every execution block is compared to the current global time. If the send instant is reached, the message is sent and the next send instant is set.

In case of a TT receiver, the data in the port is read, while an ET receiver checks the queue and, if available, receives messages from the queue. Every operation is stored in a record data structure as defined in listing 3.1 3. A record contains the number of the current test run, a global address of the job in the form u.c:s.n.j (cluster, component, subsystem, network and job), its link and port, message ID and length as well as the timestamps for send or receive operations and sequence number.

```
1  typedef struct record_struct {
```

```
2
3     unsigned short testrun;
4     unsigned short cluster;
5     unsigned short component;
6     unsigned short subsystem;
7     unsigned short vn;
8     unsigned short job;
9     unsigned short link;
10    unsigned short port;
11    unsigned char  msg_len;
12    unsigned int    msg_id;
13    unsigned int   sequence;
14    gt64 send_instant;
15    gt64 rcv_instant;
16
17  } t_record;
```

Listing 5.4: Type definition of Records

End of Test Run

After a test run is over, the jobs stop communication and write the recorded data into file in CSV format on a server via NFS. The duration of a test run can be predefined in a global header file.

5.3.6 Analysis Tool

The control application mentioned explained in section 5.3.1 is responsible for the course of the test. It collects the records after every test run and restarts the cluster. After the last test run, the analysis tool is invoked in order to generate the statistics of the whole test.

Collection of Records

When a test run is over, the records generated by the application jobs are now available in a specific folder on the DECOS server. The control application establishes an SSH connection to the server, compresses all the files into a zip archive and transfers the archive via SCP to the local host.

Cluster Restart

After the records of the expired test run are saved, the cluster can be restarted for the execution of the next test run. This is done via the remote controlled power switch. The total number of test runs for the current test case is part of the XML specifications.

Analysis of Test Results

When the last test run has expired and the last zip archive is copied from the server, the user can execute the analysis tool to generate the statistics for the current test. The parameters for the analyzer are the message ID and the participating communication partners. The results encompass the minimum, maximum and average latencies in μs, the effective bandwidth in kilobit per second (kbps), and the number of discontinuities (loss of messages) and are stored in the CSV format for further processing in a database or spreadsheet application.

5.3.7 Monitoring Access

There are two different ways to interface with the DECOS cluster. Besides the NFS connection between the cluster and the server, the serial interface provides monitoring access to the target nodes of the cluster. Following connection options have to be set:

- Baud rate: 19.200 bps
- Data bits: 8
- Parity: none
- Stop bits: 1
- Flow control: none

The serial interface provides a minimal shell at the target nodes and can also be used by the application jobs to print out messages. The printing of messages is not useful during a test run, because the print operations are an unnecessary load for the CPU, but the gateways use the serial interface to print out the summaries after a test run expires. The summary contains the total number of messages handled by the gateway and opposes valid messages that were forwarded to the target network and invalid messages that are filtered out at the gateway.

Chapter 6

Experimental Evaluation

This chapter deals with the experimental evaluation of our implementation. We begin with a definition of hypotheses for our experiments and then describe a set of test cases to validate those hypothesis. Section 6.3 presents the results of the experiments which are then interpreted in section 6.4.

6.1 Hypotheses for Experiments

The hypotheses for our experiments can be categorized into two groups. The first group is dedicated to temporal partitioning and makes assumptions about timing message failures while the second part concerns spatial partitioning and naming message failures.

6.1.1 Temporal Partitioning

Temporal partitioning guarantees that the behavior of a job in one DAS does not corrupt the temporal properties (i. e. latencies of message transmissions, bandwidth of communication channels) of another job in the same DAS or the temporal properties of the service of another DAS (independent or separated by a gateway).

The gateways of the experimental implementation can enforce specific transmission patterns on the communication channel between two different DASs. The adjustable parameters are as follows:

- **Inter-arrival times**
  The gateway maintains a minimum gap between two successive message transfers, called inter-arrival time.

- **Inter-arrival times with burst**
  The gateway can be operated in a so called *burst mode* with successive periods of *activation time* and *silence time* in order to perform traffic shaping and regulating the bandwidth across two DASs. Messages are only accepted during activation time, any other messages are discarded.

- **Periodic time-triggered behavior**
  If the destination DAS follows the time-triggered paradigm, the gateway has to suit to the time-triggered schedule of that network. Therefore it has to generate a state message in every round regardless of the timing pattern of the sending job.

Considering those parameters we can make following assumptions:

Hypothesis 1: The gateway maintains a minimum gap between two successive message transfers.

Hypothesis 2: The gateway can perform traffic shaping and bandwidth regulation.

Hypothesis 3: The gateway resolves temporal property mismatches.

6.1.2 Spatial Partitioning

The spatial partitioning mechanisms guarantee data integrity between two DASs, i. e. if a faulty job corrupts a message of another job in the same DAS, the gateway prevents spatial interference with another DAS by making use of the following naming information:

- **Message identifier**
  Every message has its own dedicated identifier included in the message header. The identifier designates the content of the message and is also used for bus arbitration in some protocols (e. g. CAN, [Bos99]). The gateway is able to filter out messages with incorrect identifiers by applying a mask to the message header. Only valid messages are forwarded, while invalid messages are discarded.

- **Source address**
  Some communication protocols use a source address in the message header which identifies the sender of the message. This address can either be logical (i. e. IP) and/or physical. The gateway checks the source address of incoming messages and filters out messages from unintended senders.

- **Destination address**
  The destination address identifies the intended receiver of a message. It is either logical (i. e. IP) and/or physical and is included in the message header.

  We can make the following assumptions:

  Hypothesis 4: The gateway is able to handle naming message failures.

  Hypothesis 5: The gateway has the ability of selective redirection of messages.

6.2 Experiments

This section describes the various experiments that have been executed in order to evaluate the error containment and selective redirecting mechanisms of the exemplary virtual network and gateway setup according to the fault hypothesis of the previous section 6.1. The tables in this section show a brief description of every experiment, the assumed erroneous scenario and the how the gateway handles the occurrence of errors.

6.2.1 Temporal Partitioning

The first blocks of experiments deals with message timing failures, i. e. violation of minimum inter-arrival times, as well as different settings for the *burst mode*. The experiments are broken down into a set of tests where the sender is located in an event-triggered network and another set with a sender following the time-triggered paradigm.

Event-triggered sender

The experiments covering communication with a sender in an event-triggered network can be divided into three blocks. First, there is unidirectional communication with an event-triggered receiver, second, unidirectional communication with a time-triggered receiver and last, bidirectional communication in a client/server (request/reply) architecture, where the gateway acts as a server and replies to requests from an event-triggered job.

Table 6.1 lists the experiments created for unidirectional communication with solely event-triggered communication partners. The upper half shows the case

Test case description	Temporal Behavior	
	Scenario	Detection/Handling
ET sender sends exactly one ET message per activation interval; The Gateway does not support bursts and handles exactly one message per activation interval; The Receiver records message receptions	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages exceeding bandwidth limit at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected
ET sender sends multiple ET messages per activation interval (burst); The Gateway operates in burst mode and handles messages according to parameter settings; The Receiver records message receptions	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages violating burst timing pattern at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected

Table 6.1: Exemplary timing message failures for unidirectional event-triggered communication without bursts

where only one message is sent per activation interval while the lower half is dedicated to the gateway operating in burst mode. In burst mode, the bandwidth is not only limited by buffer sizes and capacities of the communication channel but also by the parameter settings of the gateway. When not in burst mode, the parameter setting is as follows: BURST_COUNT = 1, BURST_TIME = 10 ms, SILENCE_TIME = 0 ms.

If the gateway connects an ET and a TT network, exactly one message per activation interval is sent on the target network. If more than one message arrives at the gateway, the Feeder NWA updates the value in place and overwrites the old value in the gateway repository. The outgoing message only contains the last pushed value. As before, the experiments listed in table 6.2 cover cases where the gateway does not support burst and where the gateway is operating in burst mode.

Experiments for evaluating temporal partitioning mechanisms for bidirectional data flow are shown in tables 6.3 and 6.4. An event-triggered client sends request messages and expects reply messages containing the required information. There

Test case description	Temporal Behavior	
	Scenario	Detection/Handling
ET sender sends exactly one ET message per activation interval; The Gateway does not support bursts and handles exactly one message per activation interval; The TT receiver records message receptions	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages exceeding bandwidth limit at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected
ET sender sends multiple ET messages per activation interval (burst); The Gateway operates in burst mode and handles messages according to parameter settings; The TT receiver records message receptions	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages violating burst timing pattern at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected

Table 6.2: Exemplary timing message failures for unidirectional ET to TT communication

are differences in the gateway automata regarding the paradigm of the network providing the required information. Whereas the ET to ET gateway automaton only acts as a forwarder of messages, the ET to TT gateway automaton needs to read the desired state information from the TT network and compute and send an ET reply message to the requesting job.

Time-triggered sender

Table 6.5 list experiments for evaluating temporal partitioning with a TT sender and either and either an ET or a TT receiver. There are differences in the layout of the gateway repository and the retrieval NWA since ET messages have to be buffered and state variables are updated in place.

Test case description	Temporal Behavior	
	Scenario	Detection/Handling
ET client sends exactly one request message per activation interval; The Gateway does not support bursts and forwards one request message per activation interval to the target network; ET server is triggered by requests and answers with reply messages	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages exceeding bandwidth limit at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected
ET client sends multiple request messages per activation interval (burst); The Gateway operates in burst mode and forwards request messages according to parameter settings; ET server is triggered by requests and answers with reply messages	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages violating burst timing pattern at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected

Table 6.3: Exemplary timing message failures for bidirectional ET to ET communication

6.2.2 Spatial Partitioning

The second block of experiments is intended to evaluate the gateways' spatial partitioning mechanisms. The tests encompass invalid source addresses, destination addresses, and identifiers according to the fault hypothesis. The partitioning mechanisms are implemented in the Feeder NWA where the selective redirection of messages takes place.

Event-triggered sender

Table 6.6 lists test cases for the evaluation of spatial partitioning for unidirectional message flow. The Feeder NWA checks message lengths, identifiers (for CAN) and source and destination addresses (for IP). If a mismatch is detected, the message

Test case description	Temporal Behavior	
	Scenario	Detection/Handling
ET client sends exactly one request message per activation interval; The Gateway acts as server and answers to one request per round from the ET network; On the TT side, the gateway automaton receives TT state messages and updates the variable in the repository	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages exceeding bandwidth limit at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected
ET client sends multiple request messages per activation interval (burst); The Gateway acts as server (in burst mode) and answers requests according to parameter settings. On the TT side, the gateway automaton receives TT state messages and updates the variable in the repository	Minimum interarrival time $<$ 100% of TDMA round	Gateway, Receiver; Gateway discards messages violating burst timing pattern at Feeder NWA; Receiver observes message loss
	Minimum interarrival time $\geq$ 100% of TDMA round	correct behavior expected

Table 6.4: Exemplary timing message failures for bidirectional ET to TT communication

will be discarded with a notification on the gateway node.

Table 6.7 lists test cases for the evaluation of spatial partitioning for bidirectional message flow in a client/server architecture. All communication partners make validity checks on the message.

Time-triggered sender

There is no need for making validity checks on time-triggered messages, because a time-triggered message does not contain any meta information concerning its contents.

Test case description	Temporal Behavior	
	Scenario	Detection/Handling
TT sender sends periodically; The Gateway simply forwards messages according to paradigm of the target network; Receiver records message receptions	Update less than once per round	Gateway, Receiver; Receiver observes duplicate sequence numbers
	Update exactly once per round	correct behavior expected
	Update more than once per round	Gateway, Receiver; Receiver observes loss of sequence numbers

Table 6.5: Exemplary timing message failures for unidirectional time-triggered communication

Test case description	Naming	
	Scenario	Detection/Handling
ET sender; Gateway makes validity check in Feeder NWA and forwards valid msgs; ET receiver checks message and records reception	invalid CAN identifier	Enabled identifier set
	invalid IP destination address	Enabled destination addresses
	invalid IP source address	Enabled source addresses
ET sender; Gateway makes validity check in Feeder NWA and forwards valid msgs; TT receiver checks message and records reception	invalid CAN identifier	Enabled identifier set
	invalid IP destination address	Enabled destination addresses
	invalid IP source address	Enabled source addresses

Table 6.6: Exemplary naming message failures with unidirectional event-triggered communication

6.3 Results

This section covers the results of the experiments discussed earlier (cf. section 6.2). The first part is dedicated to temporal partitioning. The analysis concentrates on the latencies, bandwidths and discontinuities of the communication over the

Test case description	Naming	
	Fault	Detection
ET client sends request messages; Gateway makes validity check in Feeder NWA and forwards valid msgs; ET server checks message, generates reply and records reception	invalid CAN identifier	Enabled identifier set
	invalid IP destination address	Enabled destination addresses
	invalid IP source address	Enabled source addresses
ET client sends request messages; Gateway acts as server, checks validity in Feeder NWA and generates reply message from TT state variable in repository	invalid CAN identifier	Enabled identifier set
	invalid IP destination address	Enabled destination addresses
	invalid IP source address	Enabled source addresses

Table 6.7: Exemplary naming message failures with bidirectional event-triggered communication

Network	Sender Job / Component	Receiver Job / Component	Associated Tests
Navigation DAS	job2_navigation / SCU1	job4_navigatoin / SCU0	used for tests with an ET sender
Lights DAS	job2_lights / XCU3	job4_lights / XCU4	used for tests with an ET sender

Table 6.8: Independent communication partners

gateway. The figures also show the communication of two jobs in a network that is independent from the networks connected by the gateway. Throughout the experiments it can be seen that the gateways do not affect the communication in the independent network.

The communication partners in the independent network are listed in Table 6.8.

The second part of this section presents the result of the test evaluating spatial partitioning. The tables show the number of faulty messages sent vs. the number of messages filtered out by the gateway.

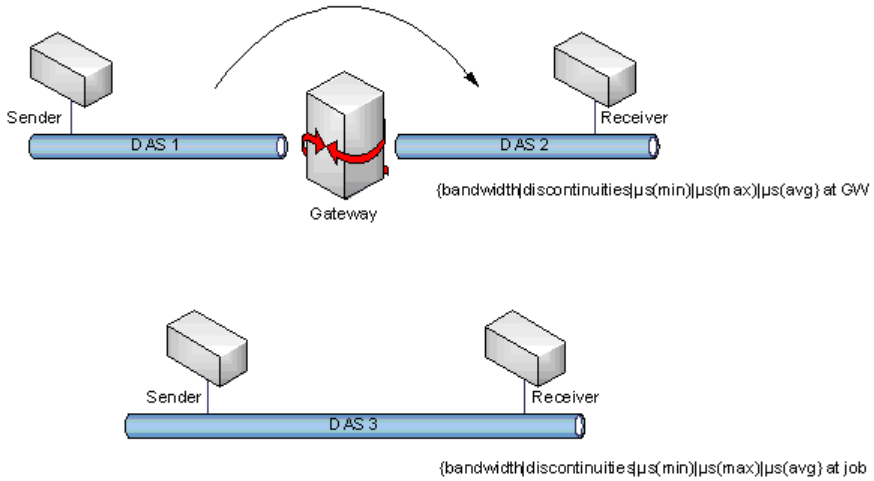


Figure 6.1: Denotation [OHP07]

6.3.1 Temporal Partitioning

As discussed in section 6.1, temporal partitioning is concerned with the containment of the effects of a timing message failure as introduced in the fault model. The virtual or physical gateway preserves the predefined temporal properties (i. e. bandwidths, latencies, variability of latencies) of the interconnected networks despite the redirection of messages. Consequently, a job connected to one network cannot affect the temporal properties of messages transmitted by jobs on another networks.

Figure 6.1 illustrates the denotation used in the result figures of this section. The upper part of the figure shows the communication over the gateway, the results resolving from this communication are denoted as *bandwidth, discontinuities,* $\mu s(min)$*,* $\mu s(max)$*,* $\mu s(avg)$ *at GW*. The reference jobs' results are denoted as *bandwidth, discontinuities,* $\mu s(min)$*,* $\mu s(max)$*,* $\mu s(avg)$ *at job*.

Two types of figures are used in this section, bandwidth vs. discontinuities diagrams and latency diagrams:

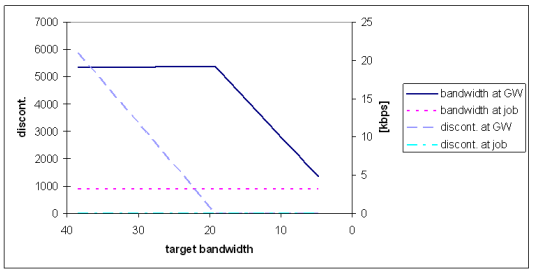
- **Bandwidth vs. discontinuities**
  The x-axis is assigned to the target bandwidth (i. e. the bandwidth at which the sender tries to send) in kilobit per second (kbps). The primary y-axis (right hand side) shows the effective bandwidth observed at the receiver (and thus achieved by the gateway), while the secondary y-axis (left hand side) shows the discontinuities observed at the receiver.

- **Latencies**
  The x-axis shows the target bandwidth of the sender while the y-axis is

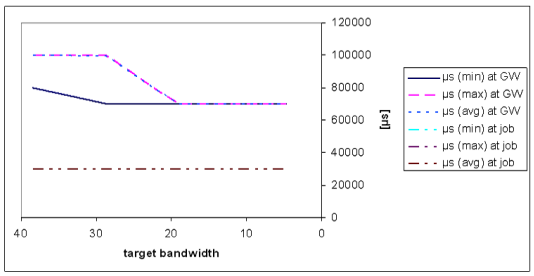
assigned to the minimum, maximum and average latencies of message transmissions.

Event-triggered periodic tests

In the first test, the gateway is configured to allow 2 messages per activation (BURST_COUNT = 2, BURST_TIME = 10 ms, SILENCE_TIME = 0 ms). This is the maximum number of messages that can be achieved by the communication channel and limits the bandwidth to 19.2 kbps. If the sender tries to send more messages per round, the number of discontinuities (loss of messages) increases because of queue overflows at the sender. The latencies, especially the maximum latency will also increase significantly because messages have to be buffered. When the target bandwidth is below or equal to 19.2 kbps, everything works as intended, i. e. the effective bandwidth matches the target bandwidth, the latencies remain constant at 70 ms (7 TDMA rounds) and no messages are lost. This behavior can be seen in figure 6.2(a) and 6.2(b).



(a) Bandwidth vs. discont.

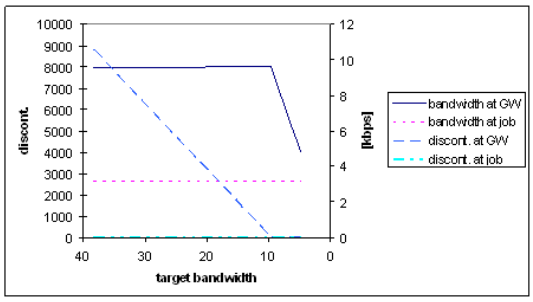


(b) Latencies

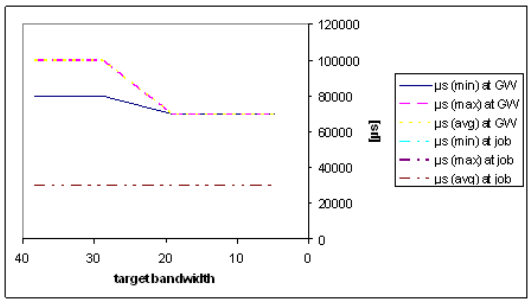
Figure 6.2: ET1_to_ET2, BC = 2, BT = 10 ms, ST = 0 ms

In the second case, the same gateway is tested with *BURST_COUNT = 1*, which means that the gateway allows only one message per burst (one round in this case). This limits the effective bandwidth to 9.6 kbps and results in discontinuities at a target bandwidth above 9.6 kbps. Like in the first test case, the latencies increase at bandwidths above 19.2 kbps, because the communication channel is not designed to support higher bandwidths. The results are depicted in figure 6.3(a) and 6.3(b).

Figure 6.4(a) and figure 6.4(b) show the behavior of the ET1 to ET2 gateway working in burst mode (BURST_COUNT = 4, BURST_TIME 20 ms, SILENCE_TIME 10 ms), which means that there are alternating periods of burst and silence time. The gateway only accepts messages in the burst period and rejects discards messages in the silence state. In the first case (cf. figure on left hand side), the sender supports the burst mode and does not send messages in silence time. Thus there



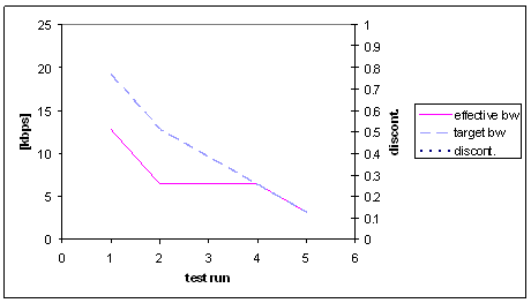
(a) Bandwidth vs. discont.



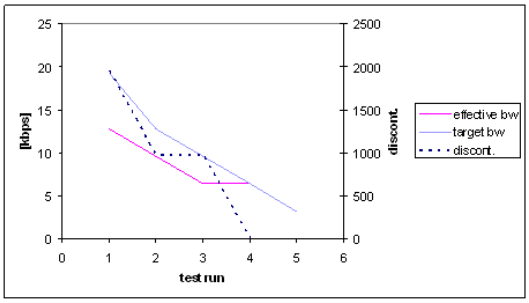
(b) Latencies

Figure 6.3: ET1_to_ET2, BC = 1, BT = 10 ms, ST = 0 ms

are no discontinuities, but the effective bandwidth is below the theoretical target bandwidth most of the time. In the second case, the sender is not aware of the burst mode and sends messages according to the target bandwidth. The effective bandwidth is again below the target bandwidth. Since the sender does not maintain the silence periods, the results in Figure 6.1 6 show a significantly large number of discontinuities. This leads us to the conclusion, that the burst mode with dedicated patterns of burst/silence periods is only useful, when sender and gateway are adjusted to the same pattern. Figures 6.4(a) and 6.4(b) use a different denotation. They show the deviation of the effective bandwidth from the target bandwidth and the discontinuities along the executed test runs.



(a) Sender supports burst mode

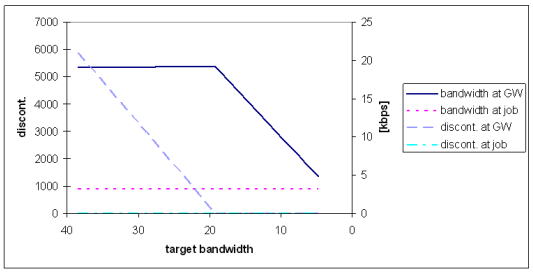


(b) Sender doesn't support burst

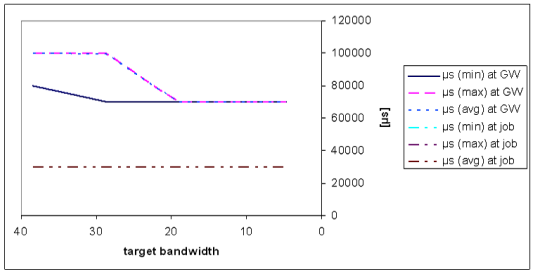
Figure 6.4: ET1_to_ET2, BC = 4, BT = 20 ms, ST = 10 ms

The results of the tests for temporal partitioning at the Request/Reply gateway look exactly the same as the results of the ET1 to ET2 gateway. This is no surprise since the temporal partitioning mechanisms implemented at the feeder network adaptor are the same. Figures 6.5(a) and 6.5(b) depict the results when the gateway allows two messages per round. In the test depicted in Figure 6.6(a) and figure 6.6(b) the gateway limits the bandwidth at 9.6 kbps by only accepting

one message per activation. Again, the latencies remain constant below a sender bandwidth of 19.2 kbps.

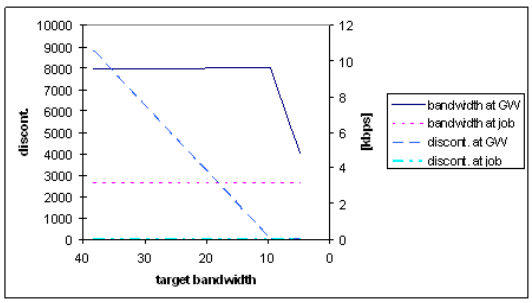


(a) Bandwidth vs. discont.

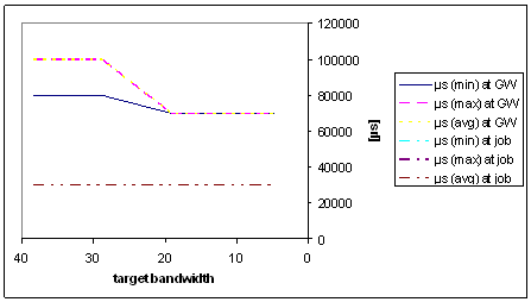


(b) Latencies

Figure 6.5: Request/Reply, BC = 2, BT = 10 ms, ST = 0 ms



(a) Bandwidth vs. discont.



(b) Latencies

Figure 6.6: Request/Reply, BC = 1, BT = 10 ms, ST = 0 ms

Figure 6.3.1 depicts the behavior of the ET to TT gateway. It can be seen that if the sender sends more then one message per round (at a bandwidth above 9.6 kbps), messages are lost. This is because in the TT network, exactly one message is sent per round. The most conspicuous characteristic of the figure is that the maximum latency increases in steps at low bandwidths. When the sender sends less than one message per round, it happens, that the TT receiver reads the same value in two consecutive TDMA rounds, thus the maximum latency increases exactly by 10 ms. The next step occurs when the ET job sends less than one message in two rounds, so the receiver reads the same value trice.

Event-triggered sporadic tests

In the first case, messages are sent with a fixed delay of 15 ms and a random delay from 10 ms to 100 ms. The diagram in figure 6.3.1 shows that the bandwidth

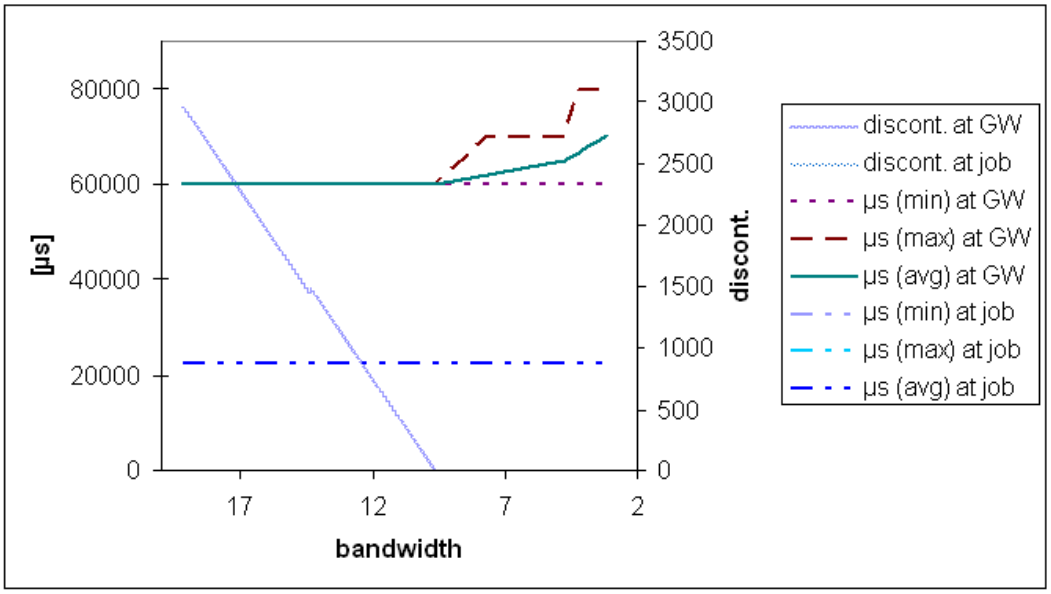


Figure 6.7: ET1_to_TT1, bandwidth vs. latencies and discontinuities

decreases with increasing random delay, while the latencies remain constant at 70 ms (7 TDMA rounds). In the second case, 10 test runs are executed with the same parameters, 15 ms fixed delay and 10 ms random delay. One should expect that the effective bandwidth varies around the expectancy of 4.8 kbps, but the bandwidth remains stable at 4.78 kbps because of the weak random number generator in C. But what's really important is, that the latencies are stable again at 70 ms (cf. figure 6.3.1). The figures use a different denotation, because the variable parameter is the random delay instead of the target bandwidth.

Time-triggered periodic tests

The results for the periodic tests of the TT1 to TT2 gateway can be seen in figures 6.10(a) and 6.10(b). In the figure on the left hand side, the sender job updates the state variable once per round, which leads to constant latencies of 57.5 ms and no discontinuities (i. e. no loss of messages). In figure 6.10(b) on the right hand side the sender works at 50% update rate, which means that the state variable is updated once in two rounds (every 20 ms). Since TT messages are sent across the network in every round anyway, the receiver gets every message twice, once after 57.5 ms and once after 67.5 ms, exactly on TDMA round later. This leads to an average latency of 62.5 ms. The bandwidths remain stable at 3.2 kbps in both cases.

The TT1 to ET1 gateway has been tested in three different ways. In the first

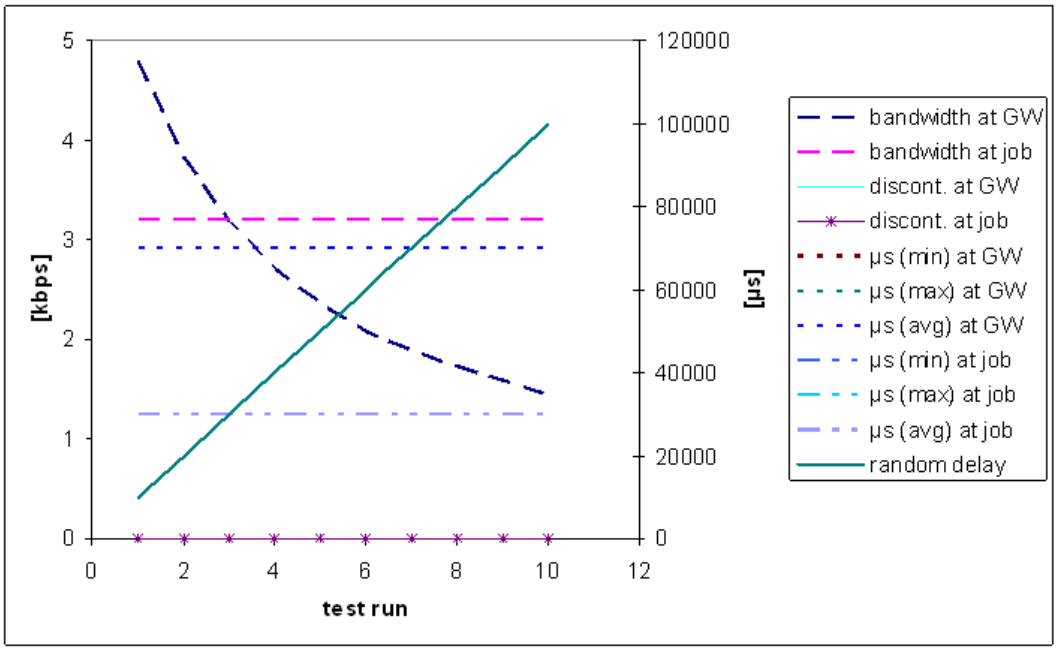


Figure 6.8: ET1_to_ET2, bandwidth and latencies, increasing random delay

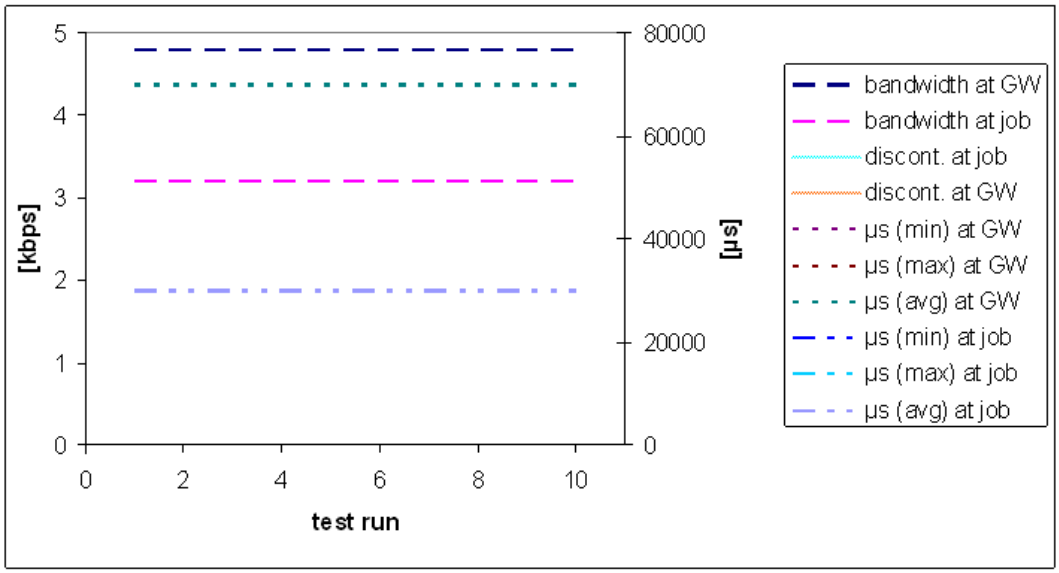
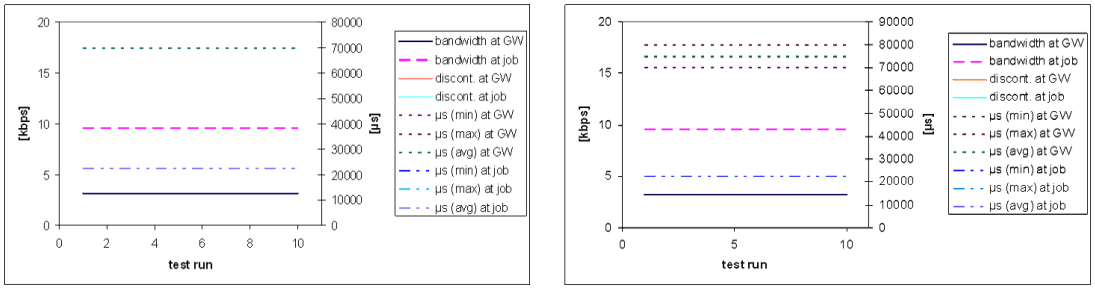


Figure 6.9: ET1_to_ET2, bandwidth and latencies, constant random delay

case, the TT sender works at 100% update rate, which means, the state variable is updated in every TDMA round and the gateway has to send an ET message in every round too. The results are depicted in figure 6.3.1. Since TT messages are 4 bytes long and ET messages are 12 bytes long, the bandwidth on the ET side is



(a) Update rate 100% (b) Update rate 50%

Figure 6.10: TT1_to_TT2

three times higher (3.2 vs. 9.6 kbps). The latencies remain constant at 67.5 ms throughout the test runs.

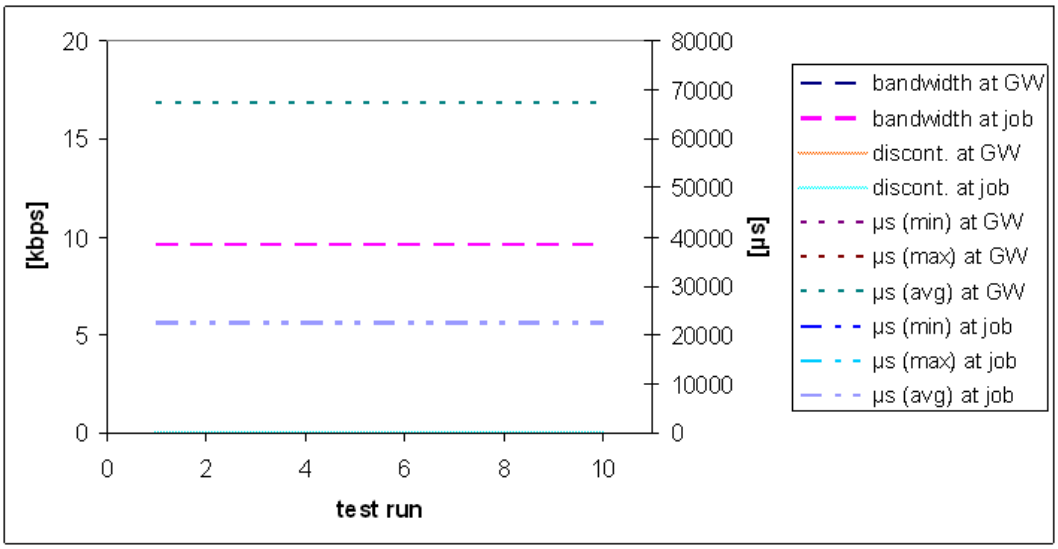
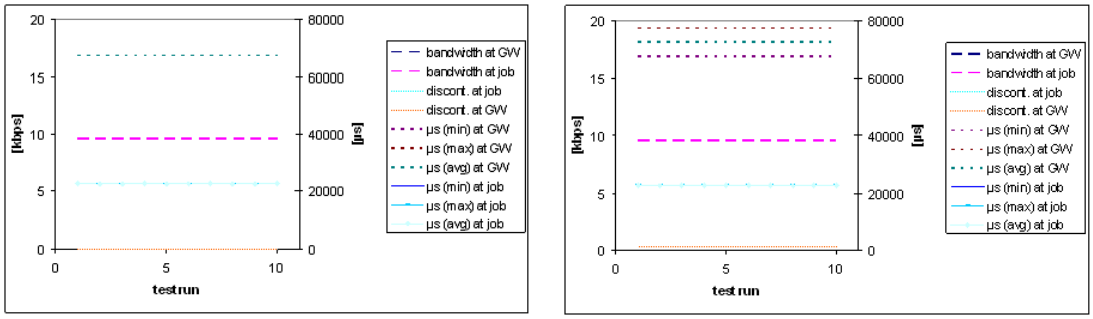


Figure 6.11: TT1_to_ET1, update rate 100%

In the second case the TT sender works at 50% update rate and the gateway is configured to forward a message only when the convertible element (the state variable in the TT message) has changed. This leads to a lower bandwidth of 4.8 kbps at the receiver, to constant latencies at 67.5 ms and no discontinuities. This behavior is depicted in figure 6.12(a). Figure 6.12(b) on the right hand side shows the behavior when the gateway is configured to forward every message, even though the convertible element does not change. This leads to a higher bandwidth of 9.6 kbps on the ET side and to a higher maximum latency of 77.5 ms, because the receiver receives every message twice, one TDMA round shifted. This behavior does not comply with the ET behavior, so the duplicate messages are counted as

discontinuities.



(a) GW forwards when variable changes (b) GW forwards periodically

Figure 6.12: TT1_to_ET1, update rate 50%

Time-triggered sporadic tests

Sporadic tests are not necessary for the TT paradigm, because messages are sent at a priori specified global points in time, in our case, once per TDMA round.

6.3.2 Spatial Partitioning

The spatial partitioning mechanisms of the gateways are implemented in the Feeder NWAs. If a message with an invalid identifier arrives at the gateway, the message will be consumed from the queue but no data will be forwarded (i. e. pushed into the repository and sent by the Retrieval NWA). The tables in this section show a summary of the valid and invalid messages sent on the feeder network. An invalid message is a message with an invalid CAN identifier or an invalid IP address. The number of invalid messages ranges from 100 to 0% in steps of 20%. The sender is configured to send at 100% (one message per TDMA round) throughout the tests.

Event-triggered sender

Table 6.9 shows the results for the spatial partitioning tests of the ET1 to ET2 gateway. The sender works with 9.6 kbps, i. e. one 12 byte message per TDMA round. The table shows the percentage of invalid messages sent over the network, reaching from 100 to 0% in steps of 20%. It can be seen that the numbers of valid and invalid messages of the sender and the gateway match, so the gateway recognizes all invalid messages and discards them. Thus the gateway works correctly.

The Request/Reply gateway that connects a TT and an ET network receives request messages from the ET network, checks its CAN identifiers sends a reply

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.9: GW ET1_to_ET2, sender at 9.6 kbps

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.10: GW REQUEST/REPLY, sender at 9.6 kbps

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.11: GW ET1_to_TT1, sender at 9.6 kbps

message, containing the sequence number of the request message and the state variable of the TT message, back to the requesting job. There is no check necessary for the T message. Since the partitioning mechanisms for this gateway do not differ from those of the ET to ET gateway, it is not surprising that the tests show the same results. This can be seen in Table 6.10.

The next table, Table 6.11, shows the test results for the ET to TT gateway. The Feeder NWA is the same for the ET to TT and the ET to ET gateway. This results in the same output and leads us to the conclusion that the gateway correctly performs spatial partitioning.

The IP gateway is tested twice, once with messages containing invalid destination addresses and once with invalid source addresses. Since the IP header is much longer than the CAN header (20 vs. 4 bytes), the necessary bandwidth is signifi-

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.12: GW IP, sender at 22.4 kbps, invalid Destination Address

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.13: GW IP, sender at 22.4 kbps, invalid Source Address

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.14: GW IP2CAN, sender at 22.4 kbps, invalid Source Address

cantly higher. Again, one message per TDMA round is sent, which leads us to the same results as before in both cases. The test results can be found in Table 6.12 and Table 6.13.

The results for the IP2CAN and the CAN2IP gateway are shown in Table 6.14 and Table 6.15.

Time-triggered sender

Since time-triggered messages only contain the state variable itself and no additional information (like a message header), there is no need to perform checks for spatial correctness upon them.

invalid messages in percent	100%	80%	60%	40%	20%	0%
valid messages sent	0	588	1176	1764	2352	2940
invalid messages sent	2940	2352	1764	1176	588	0
overall sent messages	2940	2940	2940	2940	2940	2940
valid messages received at gw	0	588	1176	1764	2352	2940
invalid messages received at gw	2940	2352	1764	1176	588	0
overall received messages at gw	2940	2940	2940	2940	2940	2940

Table 6.15: GW CAN2IP, sender at 9.6 kbps, invalid CAN identifier

6.4 Interpretation

This section summarizes the results of the experimental evaluation and interprets the satisfactory outcomes regarding the hypotheses proposed in section 6.1.

6.4.1 Temporal Partitioning

To demonstrate partitioning it is necessary that the communication latencies and communication jitter for messages sent by one job are independent from the communication activities of other jobs. In the same manner the gateway should not affect the communication resources of the jobs of the component. Concerning the results in section 6.3, where the gateway communication and the communication of an independent job in another virtual network than the feeder or retriever network is shown, the conclusion can be drawn, that the gateway service does not affect the bandwidths and latencies of other virtual networks. The test cases were performed with maximum payload on every virtual network except the networks used for the gateway (parametric variation of the bandwidth over the gateway). The latencies of the reference networks are, depending on the physical location of the reference jobs, at 22.5 and 30 ms. Throughout the whole test period there was no noticeable impact on the reference communication structure. Thus the conclusion can be drawn that the gateway service is executed independently and does not block resources needed by other services.

The gateway can enforce the temporal behavior of the feeder network with the introduced variables BURST_COUNT, BURST_TIME and SILENCE_TIME. Therefore the sender jobs need to know the settings of the gateway to be able to send timely messages. These configuration parameters make it possible to limit the bandwidth (in bits per second) of the feeder network to a desired value, which can be calculated by following formula:

$$bandwidth(bc, bt, st, rt, ml) = \frac{\frac{bc}{bt} * ml * 8}{rt} * \frac{bt}{bt + st}$$

Abbreviation	Name	Unit	
bc	BURST_COUNT	Messages	[1]
bt	BURST_TIME	Rounds	[1]
st	SILENCE_TIME	Rounds	[1]
rt	ROUND_TIME	Seconds	[s]
ml	MESSAGE_LENGTH	Bytes	[B]

Table 6.16: Parameters for bandwidth formula

$\frac{bc}{bt}$ is formed by the virtual network service as cause of the limited buffer size of the outgoing port queue. This value equals 3 messages per round in the TTP based cluster. The parameters are explained in table 6.16.

6.4.2 Spatial Partitioning

Masquerading is defined as the sending or receiving of messages imitating another sender without authority [CDK94]. Masquerading failures are often performed by systems that only rely on an explicit name stored in a message to identify the transported message. Therefore one single faulty component can masquerade other components, possibly overwriting correct messages from other components. For the receiver it is a major problem to detect those faulty messages, and as a result such messages can have a significant impact on the application. The tested gateway implementation is capable to detect such faults by relying on the correct behavior of the underlying virtual network service and by performing name checks on the incoming messages.

Spatial Partitioning with Explicit Message Names

The explicit name is part of the content of a message transported over a communication channel. For the different protocols implemented in the event triggered paradigm it can consist of an identifier in CAN messages, or the source and destination address in IP messages, which were both implemented and tested in our environment. The result tables in sections 6.3.1 and 6.3.2 reveal the correct behavior of the gateway concerning spatial partitioning with explicit message names. The gateway only forwards messages with specified identifier, source and destination addresses and drops invalid messages at his receiving port for preventing masquerading behavior at the port.

Spatial Partitioning with Implicit Message Names

Due to the virtual network service, each job is allocated to a dedicated input port (spatial context), which maps to a static TDMA slot of the underlying time triggered communication schedule (temporal context). Because of the allocation of time slots to components and their jobs, it is always possible to find out the identity of the communication partner. It was not possible to design test cases, which directly target on the verification of spatial partitioning with implicit message names, but there were no noticeable problems during a large period of test runs.

6.4.3 Property Transformation

In general, gateways have to deal with property mismatches at the semantic level or at the operational level [OPK05a]. A semantic mismatch occurs, when the meaning of a message is interpreted differently in the networks connected by the gateway. This is the case when the two networks use a different namespace. An operational property mismatch occurs, when the connected networks follow different control paradigms, or when different syntactic message formats are used.

The results in the previous section have shown that the gateways in the validation setup are able to resolve differences at the operational level of two networks, as well as differences at the semantic level.

6.4.4 Performance

The virtual networks support a certain amount of data per activation round (cf. formula stated earlier in this section 6.4 and table 6.17), which depends on the round time (10 ms) and message lengths.

This maximum transfer rate is not limited by the gateway in any test case, because its execution time remains below the worst case maximum execution time of the components and therefore there is enough time to handle all incoming messages without loss.

The latencies of messages sent over the gateway are dependable on the communication partners and keep between 60 to 90 milliseconds at the TTP based cluster and remain between these bounds. The minimum latency ranges from 60 to 70 milliseconds.

Network Name	Message Length	Maximum Transfer Rate
NETWORK_ET_COMFORT	28 Bytes	22.4 kbps
NETWORK_ET_LIGHTS	12 Bytes	9.6 kbps
NETWORK_TT_BYWIRE	4 Bytes	3.2 kbps
NETWORK_TT_NAVIGATION	4 Bytes	3.2 kbps

Table 6.17: Maximum transfer rate according to the configuration

Chapter 7

Conclusion

The DECOS integrated architecture tries to combine the benefits of federated and integrated systems by decomposing a large real-time system into nearly-independent subsystems. Those so called distributed application subsystems (DASs) are integrated within a single distributed computer system which considerably helps reducing costs.

On top of the time-triggered core architecture, virtual networks are implemented, that provide the communication infrastructure for the different DASs. It is the task of a gateway to interconnect DASs if they are required to communicate among each other. Gateways in the DECOS architecture are implemented as virtual hidden gateways (i. e. transparent to the application). Besides the transparent forwarding of messages from one network to another, gateways have to resolve mismatches if the interconnected networks exhibit different operational or semantic properties. The gateways also provide an encapsulation service that is responsible for error containment and the selective redirection of messages in order to reduce bandwidth costs.

The main purpose of this thesis is the validation of the services provided by the gateways by experimental evaluation. To solve this task in a convenient way, we proposed the use of a measurement framework that has the ability to monitor the system under test and is capable of automated execution of testruns. First we showed a model of a framework that is generally applicable and then presented an actual implementation for the evaluation of our exemplary DECOS cluster.

Several test cases have been created to test the gateways developed for the DECOS architecture with respect to error containment, property transformation and performance. The experimental evaluation shows that the gateways are able to handle timing and value message failures, allow the selective redirection of messages and are capable of enforcing a particular timing pattern on the exchange of messages (traffic shaping).

References

[Ade03] Astrit Ademaj. *Assessment of Error Detection Mechanisms of the Time-Triggered Architecture Using Fault Injection.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003. 5, 6, 14, 15, 35

[ALR01] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, 2001. 12, 13, 14

[AUT07] AUTOSAR. Automotive Open System Architecture, 2007. 9, 10

[BBD+00] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-time application interface. *Linux Journal*, April 2000. 40

[BCSS90] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using fiat. *IEEE Trans. Comput.*, 39(4):575–582, 1990. 6, 7

[Bos99] Robert Bosch GmbH. CAN specification. version 2.0, 1999. 19, 42, 60

[CDK94] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, concepts and design, Edition 2.* Addison-Wesley, 2 edition, 1994. 79

[Fle05] FlexRay consortium. FlexRay protocol specification. version 2.1., 2005. 27

[GZ79] Michel Gien and Hubert Zimmermann. Design principles for network interconnection. In *SIGCOMM '79: Proceedings of the sixth symposium on Data communications*, pages 109–119, New York, NY, USA, 1979. ACM Press. 21

[HÖ6] Martin Höller. Gateway generation for virtual networks in the DECOS integrated architecture. Master thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, Apr. 2006. 46, 49

[HPOS05] Bernhard Huber, Philipp Peti, Roman Obermaisser, and Christian El Salloum. Using RTAI/LXRT for partitioning in a prototype implementation of the DECOS architecture. *3rd Workshop on Intelligent Solutions in Embedded Systems (WISES'05), Hamburg, Germany*, May. 2005. 40, 41, 49

[J+02] C. Jones et al. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, December 2002. 22

[KN97] Hermann Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *6th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, July 1997. 12, 18, 21

[Kop95] Hermann Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 2–9, Chenju, Korea, Aug. 1995. 21

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. 11, 12, 13, 14, 15

[Kop98] H. Kopetz. The time-triggered architecture. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 22–29, Kyoto, Japan, April 1998. 5, 27

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973. 19

[OHP07] Roman Obermaisser, Bernhard Hofer, and Christian Pucher. Validation of gateways in the DECOS architecture. Research Report 15/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2007. 28, 30, 31, 32, 33, 68

[OP06] Roman Obermaisser and Philipp Peti. A fault hypothesis for integrated architectures. *Fourth Workshop on Intelligent Solutions in Embedded Systems - WISES06, Vienna, Austria, June 30th, 2006*, Jun. 2006. 30, 31

[OPK05a] R. Obermaisser, P. Peti, and H. Kopetz. Virtual gateways in the DECOS integrated architecture. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 2*, page 134.1, Washington, DC, USA, 2005. IEEE Computer Society. 1, 22, 23, 24, 42, 80

[OPK05b] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 241–253, Washington, DC, USA, 2005. IEEE Computer Society. 1, 20, 21, 26, 41

[OPT07] R. Obermaisser, P. Peti, and F. Tagliabo. An integrated architecture for future car generations. In *Real-Time Systems Journal*, pages 101–133. Springer, July 2007. 1, 9, 10, 21, 22, 26, 27, 41

[OSB+87] J. Ostgaard, R. Szkody, S. Benning, K. Purdy, and G. Mauersberger. Architecture specification for PAVE PILLAR avionics. Technical Report ADA188722, AIR FORCE WRIGHT AERONAUTICAL LABS WRIGHT-PATTERSON AFB OH, Jan 1987. 9, 10

[Pol96] Stefan Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Norwell, MA, USA, 1996. 27

[Rus99] John Rushby. Partitioning for Avionics Architectures: Requirements, Mechanisms and Assurance. NASA Langley Research Center, NASA Contractor Reprt CR-1999-209347, June 1999. 1, 9, 10, 17, 18, 19, 27

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available at `http://www.csl.sri.com/~rushby/abstracts/buscompare`. 21

[SFKI00] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. A framework for assessing dependability in distributed systems with lightweight fault injectors. *ipds*, 00:91, 2000. 7, 8

[SPL95] O. Sibert, P. A. Porras, and R. Lindell. The intel 80x86 processor architecture: Pitfalls for secure systems. In *SP '95: Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 211, Washington, DC, USA, 1995. IEEE Computer Society. 18

[TTT02a] TTTech Computertechnik AG. Time-triggered protocol TTP/C - high level specification document, 2002. 5, 39

[TTT02b] TTTech Computertechnik AG. TTP monitoringnode - a TTP development board for the time-triggered architecture, March 2002. 39

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM Press. 17