COMPONENT-BASED FAULT TOLERANCE FOR DISTRIBUTED REAL-TIME AND EMBEDDED SYSTEMS

By

Friedhelm Wolf

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2009

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

To Jesus Christ,
who not only laid the foundations of western civilization and research
through revealing the nature of God the father,
but also made me the person I am.

ACKNOWLEDGMENTS

I'm indebted to the GI department at the European Space Operations Centre and especially to Dr. James Eggleston, who not only greatly influenced the direction of my research but also was very supportive during my internship there and for the mission control system case-study.

I want to thank my adviser Dr. Douglas C. Schmidt for his encouragement to embark on graduate studies and his commitment to push my programming skills and critical thinking towards perfection. He is the reason I pursued my Masters degree and I'm thankful for these two years where I learned a lot, and even quite a bit about computer science.

I am grateful for all present and past members of the DOC group to create a pleasant working atmosphere through their openness in answering the questions of a newcomer.

My special thanks goes to Jaiganesh Balasubramanian, Jeff Parsons and Will Otte for all the productive and fun work on the projects we did together.

As a student from a foreign country I am very thankful for the people that reached out to me and cared for me. I especially want to thank my colleague Joe Hoffert and his family for inviting me to church and for supper and caring for my. This time also wouldn't have been the same without the amazing hospitality of family Joffrion that adopted me instead of just giving me a room.

The people that made all this possible, however, are my parents Marianne and Gerhard, who not only brought me up to persevere in hard work by being a living example but also by letting me go and giving me their blessing on this path that lead me far away from home.

TABLE OF CONTENTS

Page

DEDICATION . . . ii

ACKNOWLEDGMENTS . . . iii

LIST OF FIGURES . . . vi

Chapter

I. Introduction . . . 1

II. Background . . . 3

- II.1. Dependency Analysis for Fault Correlation . . . 3
- II.2. Frameworks for Fault-Tolerance . . . 5
- II.3. Modeling Dependability Aspects . . . 6

III. Basic Design Concepts . . . 8

- III.1. Fault Model . . . 8
- III.2. Architectural Foundations . . . 8
  - III.2.1. The FLARe Real-Time Fault-Tolerance Framework . . . 9
  - III.2.2. The CORBA Component Model . . . 11
  - III.2.3. The OMG Deployment and Configuration Specification 12

IV. Components with Heterogeneous State Synchronization . . . 15

- IV.1. Problem Statement . . . 15
- IV.2. Providing a Common Interface for Exchanging Diverse State Snapshots . . . 18
- IV.3. Satisfying varying Timing Requirements . . . 19
- IV.4. Support for Different Protocols for State Dissemination . . . 21

V. Component Replication based on Failover Units . . . 25

- V.1. Case Study . . . 25
- V.2. Requirements for Component Group Failover . . . 28
  - V.2.1. Requirement 1: Fault Isolation . . . 28
  - V.2.2. Requirement 2: Ensure Fail-Stop Behavior . . . 29
  - V.2.3. Requirement 3 : Server Recovery . . . 29
- V.3. The CORFU Architecture . . . 30
  - V.3.1. Challenge 1 - Single Component Fault-Tolerance . . . 31
  - V.3.2. Challenge 2 - Integration into the Deployment and Configuration Infrastructure . . . 35

V.3.3. Challenge 3 - Failover Ordering of Replicas 41

VI. Results 45

VI.1. Benefits of Component-based Fault-Tolerance compared to Object Level Fault-Tolerance 45
VI.2. Experimental Results 49
VI.2.1. Testbed 50
VI.2.2. Failover Latency 50
VI.2.3. Failover Unit Shutdown Latency 52
VI.2.4. Discussion 56

VII. Concluding Remarks 59

VII.1.Lessons Learned 59
VII.2.Future Work 61

REFERENCES 63

LIST OF FIGURES

Figure Page

III.1. Component Based Mission Control System 13

IV.1. Callback interface for state replication 18

IV.2. State transmission sequence based on a common interface 20

IV.3. The strategy pattern applied to state synchronization 23

V.1. Component-Based Mission Control System 26

V.2. Structural Overview of a Fault-Tolerant Component Server 34

V.3. Application of the Decorator Pattern for the FaultCorrelationManager design 38

V.4. Structure of a Fault-Tolerant Component System 39

V.5. IDL Declaration of RankList Constraints 42

VI.1. Development Obligations for Server-Side Fault-Tolerance 46

VI.2. Development Obligations for Client-Side Fault-Tolerance 47

VI.3. Experiment Setup for Failover Latency Measurement 50

VI.4. Failover Latency Measurements 52

VI.5. Experiment Setup for Failover Unit Shutdown Latency Measurement . . 54

VI.6. Measurement results for fail-stop latencies 56

CHAPTER I

INTRODUCTION

Research in systems engineering has focused on dependability for more than three decades. Due to industry standards and maturing production processes, hardware has become increasingly more reliable. Therefore software dependability has become the most crucial element for overall system dependability.

The goal of dependability research is to provide means for software development that not only ensure correct function but also reliability, security and availability [1]. As systems increase in complexity, the challenges of making a system dependable and especially fault-tolerant concurrently increase.

This is particularly true for distributed real-time and embedded (DRE) systems, such as traffic control systems, weather observation systems, total shipboard computing or highly automated assembly lines. These systems are characterized by limited resources, including space, energy consumption, memory size, CPU capacity and network bandwidth. Since they always include physical elements and require timely interaction with physical processes, they require diverse quality of service (QoS) guarantees, such as timely delivery of data, limited usage of processing resources or high availability. Fault-tolerance is one aspect of QoS requirements that gains more importance as DRE systems are used in mission critical scenarios with high dependability requirements.

A recent position paper [13] makes clear that despite many efforts in research, the main reasons for system crashes and downtime are problems related to dependability and fault-tolerance mechanisms fail to work correctly. The key to comprehensive system dependability lies in moving from point solutions for specific scenarios towards dependability engineering that integrates all aspects through well understood models, metrics, development processes and tools.

This thesis proposes the use of component-based software development techniques to improve fault-tolerance in DRE systems. Two frameworks are presented that capture different aspects of fault-tolerance. Components with HEterogeneous State Synchronization (CHESS), applies the strategy pattern to provide automated synchronization mechanisms for internal component state. CHESS integrates with the second framework that provides replication and failover capabilities on the abstraction level of components and groups of components. This framework is called COmponent Replication based on Failover Units (CORFU) and allows for standardized approach to fault-tolerance with a high level of transparency to the component developer.

The remainder of this thesis is organized as follows. Chapter **??** gives an overview of the background on research in fault-tolerance middleware. A summary of the basic principles for our work in chapter III. Based on this chapter IV presents the concepts and architecture of CHESS. CORFU as the main contribution of this thesis is motivated and described in chapter V. Chapter VI evaluates our proposed solutions qualitatively and quantitatively. Concluding remarks in chapter VII summarize the accomplished results and point out future work.

CHAPTER II

BACKGROUND

Dependability and fault-tolerance have been the focus of extensive research in application and system development. This section gives an overview of the efforts in three areas that are closely related to the work of this thesis. These areas are

1. dependency analysis for fault correlation
2. frameworks for fault-tolerance
3. modeling dependability aspects

II.1 Dependency Analysis for Fault Correlation

A major challenge for effective failure handling is to gain comprehensive knowledge about which parts of a system are affected by a system fault. Faults cannot detected directly but only through the resulting errors they cause. Pinpointing the causing fault allow reasoning about system parts that are affected by the original fault. This allows fast reaction to errors before they can cause subsequent errors in other parts of the system.

Gaining knowledge about error propagation dependencies between system elements is therefore crucial for dependable systems. This information can be used to determine which system parts will eventually be compromised. This enables comprehensive failure handling as opposed to reactive approaches that only monitoring for the basic elements of the system.

Research on fault dependencies has taken different paths to gather and apply such knowledge. These approaches can be categorized into static approaches and observation based approaches.

Viera et al present an approach [15] that automates dependency analysis in component-based systems. The *Component Based Dependency Model* allows the incorporation of

diverse types of dependencies that are categorized into intra-component dependencies that define execution and error propagation paths within one component implementation and external dependencies that defined dependencies to other component or hardware and software infrastructure elements. The strength of this approach is that it includes different sources of information about the system, such as deployment information, additional component meta-data and meta-data about component connection. This approach is static in its nature since it builds its dependency information based on meta-data. It therefore cannot react on unforeseen failures or error propagation paths.

The static approach can be applied to various domains of component models. Another example of this is event correlation [7] in the domain of event based systems, where dependencies between different event sources are used to identify the original fault.

To address the limitations of static dependency information the *Automatic Failure-Path Inference* [5] approach relies on system behavior analysis at run-time. It focuses on component based web applications implemented in Java and assumes that errors that express themselves as exceptions. Fault dependencies are captured as a directed graph, called failure-propagation map. This graph is populated through direct interaction with the system. Fault injection and monitoring of resulting component crashes is used to built up an initial graph for a system. Later this graph is corrected based on non-intrusive monitoring of the system under nominal operation. While this approach is very flexible in adopting the dependency information to the system it is limited in its support of different fault types due to its focus on exceptions and the Java programming language.

This work on dependency analysis relates to our research as it provides methodologies to define groups of depended components. These groups serve as input for the algorithms and mechanisms as described in chapter V.

II.2 Frameworks for Fault-Tolerance

In general a framework for tault-tolerance integrates different aspects of dependability. The role of frameworks mainly is to enable tault-tolerance, which includes error detection, fault diagnosis, fault isolation, error recovery and system reconfiguration. However other means of dependability, such as fault-prevention fault removal and fault forecasting benefit from frameworks as well. The coverage of different tault-tolerance means by different frameworks as well as their domain scope vary widely. To outline different existing approaches they are compare and contrasted with our solution approach here. The considered frameworks are Adaptive Quality of Service for Availability (AQuA) and JBoss with Application-Generic Recovery (JAGR).

AQuA [12], an adaptive architecture for dependable distributed objects focuses on providing redundancy for distributed objects. AQuA objects are contained in replication groups that provide a variety of replication schemes realized by a message based group communication mechanism. AQuA uses CORBA to define and implement objects, but maps them to the underlying group communication mechanism. The mapping layer includes mechanisms for error detection and failover. The Fault model includes process failures, detected through heartbeat messages and data value failures. A central dependability manager coordinates groups and manages the fault tolerance infrastructure.

AQuA supports tault-tolerance on the granularity of objects, while component-based systems often need additional levels of granularity. Components themselves can be comprised of objects and dependencies between components can result in their need to failover together. component-based frameworks go beyond the general framework approach by also defining a component life-cycle and development process that allows to formalize aspects of other dependability means like fault prevention through offline analysis or defined methodologies for fault removal and system validation.

JAGR [4] builds on a component-based infrastructure for the domain of three tier web

applications with permanent data storage. JAGR focuses on intelligent failover mechanisms based on dependency information gained through automatic failure path interference as described earlier. Its main components are a modular monitoring structure that allows to plug in different monitors for different error types. An intelligent recovery manager gathers this information and applies micro-reboots to restart parts of the system that are affected. Based on the result it can escalate the reboot scope from single components to the whole system.

In distributed real-time and embedded systems however, persistent data storage and stateless components cannot be applied in all cases due to limited storage and processing resources. Our approach will therefore take into account state replication of groups of replicated components and therefore provide failover mechanisms as a major mean for fault tolerance instead of micro-reboots.

II.3 Modeling Dependability Aspects

A component-based framework that targets DRE systems is Cadena [8]. Cadena focuses on the modeling of component behavior early in the design process based on property specifications capture high-level component information. This includes inter-dependencies to ports of other components and intra-dependencies that capture relationships between ports of the same component. Properties also capture behavioral specifications that allow reasoning of temporal behavior and control-flows within components. Based on this information, interface definitions and assembly descriptions, a system model can be constructed to allow reason on various system aspects, such as event rate assignment, traffic optimized component distribution and schedulability analysis. Cadena not only encompasses a run-time framework, but also a domain specific modeling tool suite for system modeling and a simulation environment for model verification.

MDDPro[14] focuses on modeling dependability QoS requirements more explicitly. It is designed to be a domain specific modeling language that provides an orthogonal view to

the deployment structure of a system and allows the annotation of tault-tolerance attributes to components. It introduces three concepts to explicitly model component replication:

1. Failover units annotate that a group of system entities fails if any one element of it fails. Different parameters can be defined on the group that characterize the kind of failure recovery strategy used (e.g. number of replicas, heartbeat frequency, etc.).

2. Replication groups allow to formally declare, which components replicate the same logical object. Replication groups allow to configure state synchronization policies.

3. Shared risk groups are a way to model how likely it is that a failure propagates from one processing node to other nodes. This is realized as a tree where edges represent neighboring nodes and distances in number of edges serve as a measure for how likely a failure is to propagate.

MDDPro provides placement algorithms that automatically add component replicas based on those entities above and provides model interpreters for generative programming of deployment meta-data in XML format.

Our work on run-time support for component dependability is complementary to both of the approaches of Cadena and MDDPro since it provides run-time support for the modeling concepts. Chapter V describes CORFU, a framework for failover behavior for groups of components and chapter IV describes CHESS, a framework for automated state synchronization within replication groups.

CHAPTER III

BASIC DESIGN CONCEPTS

III.1 Fault Model

The underlying fault model for the work presented here includes detection of host failures and process failures. A host is a physical unit of processing that is connected to the system through a network and has an operating system. A process is located on a specific host and performs system functionality in a separate address space. It is furthermore assumed that hosts as well as processes show fail-stop behavior. This means that any occurring error leads to immediate shutdown of the entity. Due to the resource constraints posed by DRE systems only passive replication is considered, where only one primary replica is actively processing requests, while backup replicas are activated in the case of a failover as opposed to active replication that consumes more processing and networking resources.

III.2 Architectural Foundations

The prototype for a dependable component framework is based on the OMG lightweight CORBA component model specification. However the design structure of the services and mechanisms described can be easily implemented on any other component framework with mightiness. We will therefore first give an overview of the central concepts of CCM.

For the proposed architecture we furthermore take a layered approach: As CCM builds on the CORBA specification that provides object level abstraction in a distributed system, we leverage the capabilities of earlier research on fault tolerance on CORBA objects with real time requirements. Thus the second part of this section describes the capabilities of FLARE, a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems [3]. Based on these concepts we then develop the architecture of a component based fault tolerance mechanism.

III.2.1 The FLARe Real-Time Fault-Tolerance Framework

FLARe provides “lightweight fault tolerance” for CORBA objects. It combines several concepts and services to allow the definition of replication groups per object, a mechanism for failure detection and means to allow backup replica to seamlessly take over request processing of failed objects.

FLAREs design also minimizes direct coupling with application code, so that its mechanisms provide fault tolerance as transparently as possible. FLARE entities can be categorized by their location within a system. We distinguish client side entities, server side entities and middleware services.

The ***server side*** entities enable the grouping of replica objects on different machines or processes to be treated as one logical entity. FLARE provides replication on the granularity of objects but detects process level failures since it is very unlikely that an object within a project crashes without affecting the complete process and taking it down with it. To determine process crashes, each server side application includes a separate *monitor thread* that uses a TCP/IP socket to allow a monitoring service to observe the liveliness of the process. The server also includes registration functionality for the monitoring service and a central ReplicationManager, which both are described later. FLARE also provides a generic state synchronization mechanism, that requires application to provide callback methods that can insert and extract their internal state into and from a CORBA Any type. A *state synchronization agent* in the server process is responsible for retrieving and distributing the server object state. This is done by one agent for all the objects hosted in one process. To associate hosted objects with a replicated group of objects, a server side interceptor adds a tagged component with the name of the replica object group to each IOR that belongs to a locally hosted servant.

The ***client side*** entities allow seamless failover and failure detection. Using CORBAs interoperable interceptor framework, an *interceptor for exceptions* is used to detect communication failures. If an exception is detected, the interceptor consults the second entity

deployed on the client side: The *forwarding agent* keeps an up-to-date list of all the relevant object groups of a system and can therefore pass an object reference of the next replica for the failed object. The information about the object group, the failed reference belongs to is extracted from the IOR of the object as described in the server side mechanisms of FLARE. The client interceptor then uses the CORBA LOCATION_FORWARD exception mechanism to transparently redirect the clients request to a working backup replica.

The FLARE ***middleware services*** include a *replication manager* (RM) and *host monitors*. Both services are have an IDL interface and are implemented as CORBA Object Services. The host monitors responsibility is to detect process level failures. For this purpose one host monitor service is deployed on each network node that hosts server processes. Each server process needs to register itself with the host monitor and open a socket connection that can be monitored. All host monitors register themselves with the RM and send periodic updates about the host machine status to the replication. These update messages also serve as heartbeats to allow the RM to detect if a host is unreachable. The RM itself is the central entity that keeps all the information about active replicas and their location as well as the status of all host machines. It periodically builds up-to-date lists (so called RankLists) of object references belonging to one replica group and their failover order. These lists are then sent to every client forwarding agent and every server state synchronization agent to provide them with the necessary information for failover or state synchronization respectively. The RM itself can be replicated using the same mechanism as any other server object to avoid it being a single point of failure.

FLARE allows very time efficient failovers since every client has a local copy of failover targets and can use built-in ORB features to perform failovers transparently to the application logic. The second strength of FLARE is its ability to react to changes in system performance by implementing algorithms within the RM that sort the rank lists according to available system resource at the time of failure detection. This allows to choose backup replicas on the least loaded host to take over and therefore avoid performance overloads

due to failures. By providing an abstraction for a group of replicated objects FLARE lends itself as a basis for higher-level abstractions of fault tolerance as we will describe in the following section.

III.2.2 The CORBA Component Model

The CORBA Component Model (CCM) provides a framework for software components that are reusable in different contexts, without the need to recompile or adapt them due to changes of the infrastructure or other aspects unrelated to application logic.

CCM components and ports. CCM extends the CORBA Interface Definition Language (IDL) to support the definition of components. CORBA Components can expose services through so called *ports* that are defined in IDL. Ports provide a structured way for components to interact. Ports can either provide services or indicate that the component uses the service of another component. Different port types allow for either synchronous or asynchronous communication.

The CCM container model. The process of developing CCM components is supported by code generation tools. The structure of component implementations and factories for component creation, called homes, are defined by the Component Implementation Definition Language (CIDL). A CIDL compiler generates code that integrates executor code written by component developers into the CCM run-time middleware. The actual implementation of a component is called executor and accesses the run-time through special interface. At the heart of this integration is the CCM *container*. A container provides the run-time environment for one or more component implementations and consists of the following two parts:

- Obligations that component developers must implement, such as life-cycle methods (*e.g.*, `ccm_activate(), ccm_passivate(), and ccm_remove()`), support for provided ports (*i.e.*, facets and event sinks), and configuration through attribute setter and getter methods.

- Obligations that must be implemented by the CCM middleware, such as context information that component developers can use to access middleware services, such as persistence, event notification, and fault-tolerance. Likewise, all required ports (*i.e.*, receptacles and event sources) can be accessed here.

Component servers. Components implementations are compiled into libraries and then packaged together with meta-data. On system deployment, components are loaded into a processes that provides the container interface. All customization to the specific run-time is done by configuration attributes and by selecting the appropriate component implementation. The process that hosts components is *called component server*. Component servers are started by started by deployment tools that are described in the following section.

III.2.3 The OMG Deployment and Configuration Specification

In addition to the specification for CORBA components, the OMG also specified the Deployment and Configuration (D&C) specification [9] that standardizes data structures and interfaces for component meta-data and component deployment functionality. Although it can be used to deploy CORBA components it is designed to be independent of any concrete component model and can be used to deploy other types of components.

The D&C specification is segmented, containing data models, run-time interfaces and tool specifications for the three phases of component software development, target system definition and execution of a component system.

We will focus on the data and management model for system execution.

Data Model: The central model for how a system is structured is the deployment plan. It contains information about which component *implementations* and corresponding *artifacts* are used and which component *instances* are present in the system. Each of these entities can also contain configuration properties that allows tailoring of components to the

specific deployment. The target infrastructure is represented in form of *nodes*, that represent server machines that a component instance will run on. Each instance is associated with a node to run on. As mentioned earlier components interact with each other through ports. The deployment plan captures component interdependecies through *connections*. Each connection contains two references to component ports, where one reference is point to a port that provides a service and the other reference points to a port that requires a service. Connections to interfaces outside of the current deployment plan are realized through *external references* that allow to specify a CORBA object reference URL to identify the provided or used service.

Management Model: All management entities are defined by their interfaces which contain methods and attributes. The central entity is the *ExecutionManager* which is responsible for instantiating *DomainApplication*s defined as deployment plans. Every node is represented by a *NodeManager* in the management layer. For each deployment plan it will create a *DomainApplicationManager* that is the administration interface to start and stop the application. It will split a deployment plan into partial deployment plans and each NodeManager will process these plans. Each node deployment plan will be represented by a *NodeApplicationManager* that acts on the local level as the DomainApplicationManager on the global level and allows to start and stop *NodeApplications*.

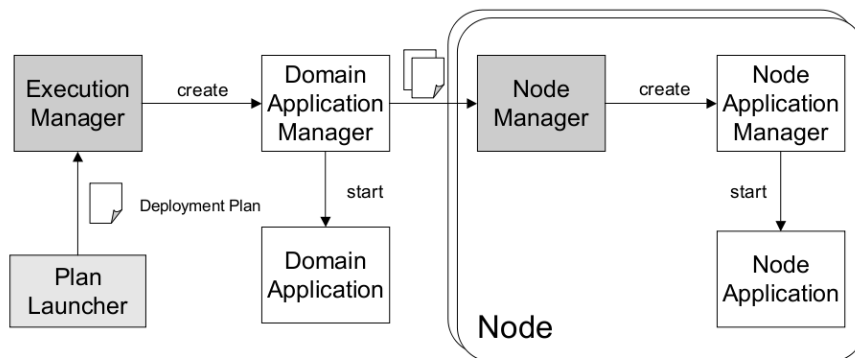

Figure III.1: Component Based Mission Control System

Figure III.1 shows the interaction patterns between management entities. The *Plan-Launcher* does not belong to the management model but is a deployment tool used to read in a deployment plan, pass it to the execution manager and guide the system start-up process.

CHAPTER IV

COMPONENTS WITH HETEROGENEOUS STATE SYNCHRONIZATION

IV.1 Problem Statement

Passive replication schemes depend on backup replicas that can take over processing quickly when a failure occurs. This includes deployment of backup instances of the same application and then failover when an error is detected. In addition to that replicas need to be synchronized frequently when they are not stateless. Applications in general and component instances in particular contain internal state. This state can change through client invocations. It is also possible that other elements in the system, such as time triggered events can modify internal state.

Active replication schemes do not necessarily need to keep replicas synchronized since all replicas process the same incoming requests and change their state accordingly. However even active replication only can ensure this in deterministic applications, where a certain input results in the same internal state every time. It also cannot be applied if state can change due to external events that are not captured by the fault-tolerance mechanism, e.g. process mutexes or shared memory as described in [11, section 3.5].

CHESS focuses on passive replication schemes. Due to changing internal state component replicas need to exchange information about their state to preserve consistency. State consistency is required for replicas to take over immediately on error occurrence. A common technique for state synchronization is the check-pointing approach: all relevant state information of an application is gathered and captured in form of a snapshot (*i.e.* structured data or memory dumps). There are different approaches for the timing on snapshots: While a time triggered approaches define an interval after which a new snapshot is taken, event triggered approaches take snapshots based on notifications from the application of system infrastructure that state changes have occurred. Depending on the replication style

snapshots are directly distributed to all replicas through dedicated communication mechanisms like multicast messages (warm passive) or they are stored in a central repository and transferred to the replica only prior to a fail-over (cold passive).

Providing a generic mechanism for state replication is a challenging task due to the wide range of differences in how application state can look like. To design such a mechanism means therefore to trade-off different characteristics of internal application state. This state can be categorized across different dimensions that will serve as criteria to evaluate the state synchronization mechanism of CHESS.

1. The **Location** of state in relation to the component implementation is a crucial aspect and limitation for application generic approaches to state replication. The most common case is state *internal* to the application, being captured in local variables, members of classes that implement the component or component attributes. However in complex DRE systems it is possible that components access system resources or middleware infrastructures (e.g. a database persistency layer) which is *external* state. A special case of external state is *shared* state where several components use a system resource (e.g. shared memory) together. Simply including external and especially shared state into the snapshot would lead to duplicates and merging conflicts in the replicas and has therefore been given careful design consideration.

2. The **Size** of the internal application state can vary greatly. On the one side of the spectrum there are *stateless* applications that have no state that needs to be preserved from invocation to invocation. Other components keep state information that is comparatively small (e.g. configuration values or counters). In other application domains state data includes large amounts of data (e.g. received streaming data, multimedia content, in-memory databases).

3. **Complexity and Distribution** are two tightly coupled properties of application state information. The term distribution tries to capture the fact that the application can

contain very different types of state that are not stored within a single data structure but rather are distributed throughout the application structure. The greater the degree of distribution the harder and more time consuming it is to create a snapshot or to restore state from a snapshot. This also applies for complexity: On the one hand there are very simple data structures like basic types that are very easily copied to or extracted from a snapshot. As the complexity increases for sequential containers like errors or lists of items, these operations get more time consuming. Associative containers and structures with arbitrary member data types and big hierarchical depth have even higher performance costs for snapshot creation.

4. **Dynamics of Changes**: Not only the form of state differs greatly from application to application, but also the frequency by which state is altered and needs to be checkpointed. Some applications alter and store their state only once at initialization. Other applications undergo many state changes in their lifetime. These changes can occur due to external input or internal mechanisms like time-triggered events. Many applications change their state based on incoming requests. Depending on the system characteristics this can happen very rarely (e.g. in applications only used for maintenance) or with a high rate of invocations in the range of microseconds (e.g. for streaming of satellite telemetry data). A generic replication mechanism like CHESS, therefore needs to offer the flexibility to specify at which timing characteristics need to be ensured for state synchronization.

We present the architecture of CHESS by presenting three design challenges that origin from the diversity of state characteristics. These challenges are:

1. providing a common interface for exchanging diverse state snapshots

2. satisfying varying timing requirements

3. support for different protocols for state dissemination

```
interface ReplicatedApplication
{
  void set_state (in any state_value);

  any get_state ();
};
```

Figure IV.1: Callback interface for state replication

IV.2 Providing a Common Interface for Exchanging Diverse State Snapshots

Challenge: As described earlier the structure and complexity of state snapshots varies greatly and in general is tightly coupled to an applications' implementation. It is therefore impossible to design an interface through which state snapshots are passed as strongly typed parameters. First generation distributed systems tended to solve this problem by passing simple byte streams and leaving the complex challenges of marshaling and demarshaling as well as type checking and alignment adaptions to the application developer.

Solution: Pass state snapshot as CORBA Any. To achieve platform and language neutrality for the state extraction mechanism and integration the necessary interfaces are declared in CORBAs interface definition language (IDL). IDL defines a special basic type `any` that allows dynamic insertion of any data type and still preserves type-safety through type code annotation and support for type checking, marshaling and demarshaling.

This allows to separate different obligations in the process of state distribution: The application itself has to perform the insertion operation of its internal state into an any object and also the extraction operation to retrieve new state instances from an any value. The middleware can then distribute the Any value transparently without needing to have additional knowledge about the internal structure of the snapshot. CORBA Anys can only contain data defined in IDL.The application developer is responsible for declaration of an IDL data type that represents the complete state, so that it can be inserted into an any data-type.

Figure IV.1 shows the obligations of an application to make its internal state available

to the state synchronization mechanism. An application has to implement these methods to interact with the state synchronization mechanism. If the framework needs to extract state from an application that is a primary replica, it will call `get_state()`. All backup replicas will receive state updates through the `set_state()` method.

Evaluation: This approaches' main strength is that it addresses the dimension of complexity and distribution by allowing the separation of concerns. It shields the generic mechanism from the internal structure of the application state but also supports the application developer by using the CORBA Any data type that provides extraction and insertion operators and therefore simplifies the gathering and composition of a state snapshot. The dimension of size has a strong influence on the performance of this approach: Transmitting any data has a certain overhead since type information has to be embedded on the sender side and extracted on the receiver side. Dealing with the location dimension of state is left to the application developer who has to solve the problem how to deal with shared state without any framework support. This particular aspect of the solution does not address the dimension of the dynamics of changes.

IV.3 Satisfying varying Timing Requirements

Challenge: Applications may have very different requirements for *when* snapshots shall be distributed from the primary replica to backup replicas. There are two main types of timing behavior: (1) cyclic timing where state is updated based on a given time interval and (2) acyclic timing where specific events like a client request trigger state synchronization. Middleware mechanisms can automatically determine when to disseminate state for cyclic timing behavior and therefore use the `get_state()` and `set_state()` methods as callback methods to automate the process. However since the timing cannot be predicted in the second case it needs active involvement of applications to disseminate state at the right time. Combining both cases into a general framework mechanism is needed to ease the burden of the application developer without restricting timing schemes.

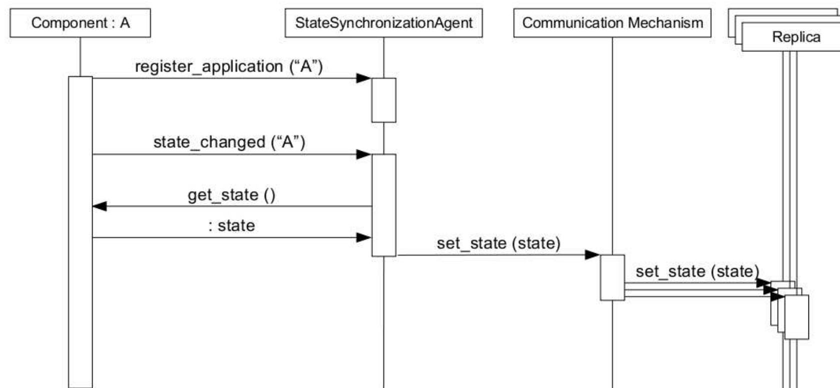

Figure IV.2: State transmission sequence based on a common interface

Solution: Separation of concerns between triggering state synchronization and state retrieval allows to treat both cases in a uniform way. This approach includes several steps of interaction between an application and a StateSynchronizationAgent which is a middleware agent for state synchronization. Each process containing CHESS object replicas also hosts a StateSynchronizationAgent that is responsible for all replication related functionality and therefore removes this obligation from the application developer.

The sequence of interactions as described in figure IV.2 provides a mechanism for flexible and generic state dissemination.

1. *Registration of components* with the StateSynchronizationAgent through a unique application id allows the manager to retrieve state from the application when needed. The registration needs to be done during the start-up phase of the component.

2. The StateSynchronizationAgent exposes the interface method `state_changed (in string id)` that allows the component to indicate a change of its internal state has. This then *triggers state synchronization*. The id parameter is needed by the agent to identify the component amongst all locally deployed components managed by this agent.

3. It is the agents responsibility to react on the notification about a state change and *retrieve the component state* from the component that issued the notification. This is done by calling back the `get_state()` method described earlier.

4. As the final step the StateSynchronizationAgent will then *distribute component state to backup replicas* in form of a CORBA Any instance.

Evaluation: This solution mainly addresses the dimension of dynamics of changes. CHESS makes triggering of state synchronization the responsibility of the application developer. The trade-off for this approach is additional effort for the developer to issue the change notifications whenever they are necessary. On the other hand this gives great flexibility in controlling which application state changes really require state synchronization. This allows for the most efficient usage of resources, since updates are only performed if they are necessary. Through the separation of concerns between state change notification and the actual execution of the state dissemination the effort for the developer is greatly reduced. CHESS shields the replica implementation from the actual distribution of snapshot data to backup replicas.

IV.4 Support for Different Protocols for State Dissemination

Challenge: There is no one-size-fits-all communication mechanism to disseminate state. Depending on size and timing requirements and the scheme of state dissemination, different communication mechanisms are needed to provide optimal performance. Small snapshots of applications with high reliability requirements need to be transferred through synchronous peer-to-peer protocols with error correction capabilities. Larger snapshots, especially when transmitted to a large number of replicas need efficient protocols like group communication protocols and multicast messages. In systems with cold passive semantics where replicas only need to update their state in a failure case a central persistent storage solution for state storage and retrieval is more adequate. Directly encoding the type of

communication mechanism into the applications' implementation results in a tight coupling between business logic and transport mechanism and therefore complicates development and adaption of the application.

Solution: Applying the Strategy pattern. CHESS uses the strategy pattern [6, pp.315f] to allow applications a flexible choice of the used protocol at run-time. The state dissemination mechanism is represented by an object interface that provides a generic way to access all variants of state dissemination in the same way. This pattern can be applied to shield the component developer from the concrete protocol for state dissemination. In this way the functionality can be integrated into the StateSynchronizationAgent. On replica registration the application can set a policy to determine which mechanism will be used by the agent. The agent then will instantiate the appropriate concrete strategy object instance and associate it with the application to use with every dissemination of state information.

Figure IV.3 shows how the strategy pattern was applied in CHESS to support two different communication mechanisms. These are synchronous CORBA calls and multicast communication based on OMGs Data Distribution Service (DDS). The design of CHESS easily allows to extend the framework by additional communication protocols, e.g. message-based mechanisms or database storage. The abstract strategy interface benefits from the earlier design decision to use the CORBA Any data type to represent snapshots. This reduces the complexity of the interface methods. However is also creates the necessity to extract the data from the any object and transform it into the appropriate form in each concrete strategy class. One example for this is shown in case of DDS communication.

The design above allows for choosing a communication mechanism choice for each replica within the process. At registration time the StateSynchronizationAgent will create the appropriate concrete strategy based on a registration parameter. When the application later notifies it about state changes the agent will pass the state to the appropriate object using the ReplicationStrategy interface.

Evaluation: CHESS flexible mechanism for heterogeneous protocols addresses the

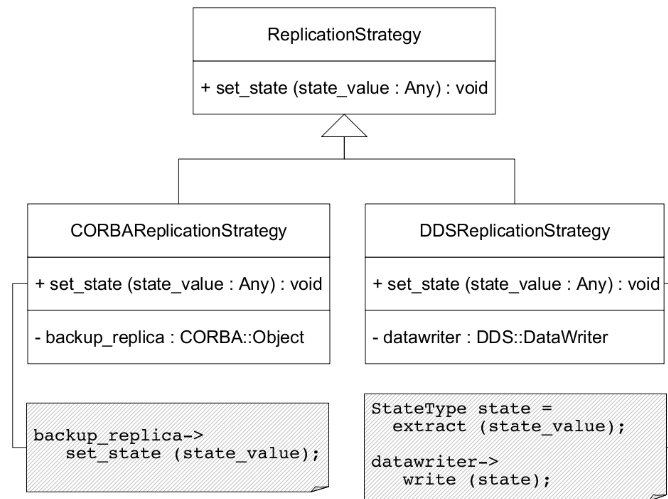

Figure IV.3: The strategy pattern applied to state synchronization

dimensions of size, complexity and change dynamics. It allows to transparently apply protocols suited for particular state characteristics. This flexibility enables trade-offs between the following aspects:

1. *Short delivery times* need to be ensured for components with high update rates where the dimension of change dynamics is important. However with growing size and complexity of state snapshots it is harder to provide short delivery times. Connection oriented protocols are well suited for fast delivery of small amounts of data.

2. *High network throughput* is necessary for snapshots with large sizes. However timely delivery can suffer from protocols that maximize throughput. Group communication mechanisms are well suited for sending large state to several receivers.

3. *Reliable delivery* is needed in systems were state consistency has to be guaranteed under all circumstances. This usually is done through error correction codes and retransmission of lost packets. Therefore trade-offs have to be made between efficient and reliable delivery protocols.

The strategy pattern allows to make these trade-offs on a per component basis and therefore accounts for heterogeneous environments and systems with highly diverse state characteristics per component.

CHAPTER V

COMPONENT REPLICATION BASED ON FAILOVER UNITS

Conventional middleware solutions provide fault tolerance through replication and recovery on the granularity level of single objects, processes and servers. Component middleware requires failover mechanisms at a higher level of granularity. The compositional nature of component applications often results in dependencies between components that require a coordinated failover mechanism for groups of components distributed across several servers and processes. This chapter presents CORFU, a middleware architecture for component-based fault-tolerance that includes support for single-component fault-tolerance and uses it as a base for providing fault-tolerance on the level of groups of components.

V.1 Case Study

The domain of space systems is one that has especially strong requirements for real-timeliness as well as for dependability. To illustrate the challenges that arise from component-based DRE systems we describe the structure of a possible Mission Control System (MCS) as used by the European Space Agency [10].

The purpose of an MCS is to control one or multiple satellites that perform a mission in space that is dedicated to a specific task, such as earth observation or deep-space exploration. A MCS processes data gathered by the satellites and controls satellites. It is deployed in a central control station and communicates with a network of ground stations that provide communication links to the satellites.

Figure V.1 shows the structure of a component-based MCS. As time windows for active connections to satellites can be very short due to their orbit and visibility to ground stations,

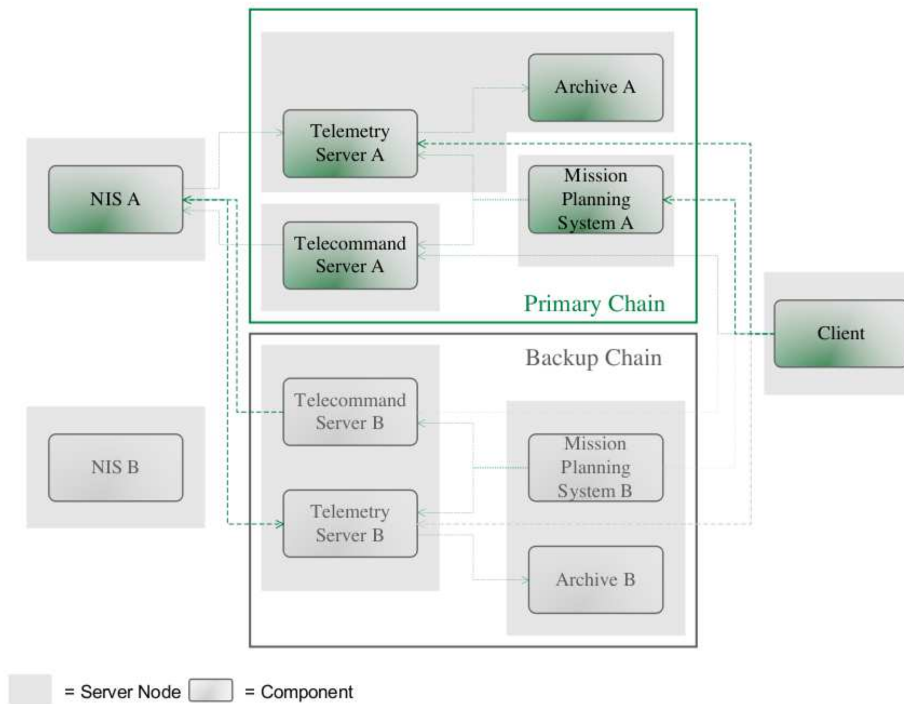

Figure V.1: Component-Based Mission Control System

availability of the MCS during such phases is crucial. All MCS are therefore laid out redundantly in hardware as well as in software functionality. Each of the entities is deployed twice and some are grouped into chains of functionality that are groups of components working closely together.

The Network Interface System (NIS) serves as a gateway from the ground stations to the MCS through a wide area network. Using a special protocol, the space link extension protocol, it processes and transmits all mission relevant data to and from the MCS. This includes sending telecommand data that controls the satellites and receiving telemetry data from the satellites. The NIS itself is not part of a MCS chain, but is laid out redundantly. Hardware and software of a NIS are tightly coupled and therefore replicated together. In case of NIS failure a chain can switch from the active NIS A to NIS B in a warm passive failover.

The telemetry server analyses telemetry data and preprocesses it for the mission operators. The archive stores telemetry data permanently and is fed by the Telemetry server. The telecommand server is responsible for creating and sending new commands issued by the mission operators. A MCS needs to be tailored to specific missions and reconfigured for different mission phases. The Mission Planning System is responsible for configuring and observing the other system entities based on the mission specific characteristics. These four entities form a MCS chain that provides the main MCS functionality. To avoid single points of failure this chain is replicated. As shown in the diagram a primary chain is active during normal operation. In case of an error within the primary chain the complete chain is passivated and a backup chain takes over operation through a warm passive failover. All components of the backup chain are already deployed to take over operation as quickly as possible. Only one chain at a time is allowed to send data through the NIS to the ground stations.

The MCS is accessed by clients that allow mission operators to interact with the system through a graphical user interface. Clients are always connected to one specific chain, usually the primary MCS chain. In case of a failure the client also needs to failover to the backup chain to ensure high availability for mission operators. Clients themselves are not replicated and can be simply restarted.

In our case study three levels of replication granularity can be found:

1. no replication as in the case of the client components

2. replication of single components as for the NIS

3. replication of groups of components as in case of the MCS chains

CORFU incorporates techniques to achieve single component replications and based on this component group replication.

V.2 Requirements for Component Group Failover

Providing a replication and recovery mechanism operating on component groups includes several requirements that need to be fulfilled. These requirements are (1) fault isolation, (2) ensure fail-stop behavior of failed groups and (3) server recovery.

V.2.1 Requirement 1: Fault Isolation

Since faults are not recognizable directly they can only be isolated through detecting occurring errors and reasoning about their cause. This then enables to predict which other parts of the system will be affected by the same fault without having to wait for other errors occurring. For component-based systems this includes determining if a failed component or a group of components within a failed process have external dependencies that allow the failure to propagate to other components. To provide fault isolation the fault tolerance mechanism needs to determine which components are affected by a failure so that actions can be taken to shield the system from this failure. This is hard since affected components are possibly deployed across several server nodes. Component dependencies exist in many forms, some being harder to detect and capture than others. They emanate from various causes, such as shared operating system infrastructure, shared use of network resources, middleware services and business logic dependencies.

Application in the MCS scenario: In the MCS scenario the reach of failure dependencies differs among components. A client component has no failure dependencies and can simply be restarted without affecting other system components. The NIS components will not require other components to restart when they crash but the telecommand server needs to be reconnected to the backup NIS. The components within one chain however are dependent on each other. It is explicitly required that a failure occurring in one of these Crash of the TM Server needs to result in marking all Chain A servers as failed. These three cases need to be treated by middleware in different ways.

V.2.2 Requirement 2: Ensure Fail-Stop Behavior

After a fault has been isolated by determining affected components it is necessary to regard these components as containing inconsistent state. This is a threat to system consistency since they possibly carry transient faults. As a consequence all affected components need to be stopped as soon as possible. The time from error detection to the complete stop of all affected components needs to be minimized. This is hard due to two factors: (1) The time needed between detection of the first error and the effective shutdown of affected components and (2) the need to synchronize the shutdowns between components in a distributed environment.

Application in the MCS scenario: If a failure has been detected in the telemetry server this could affect the other components in the chain and lead to inconsistency. The archive might store data that is not correct and the telecommand server might issue commands based on telemetry data that is no longer valid. Since the different components run on different hosts, the shutdown process cannot happen instantaneously but will be affected by the reaction time of the system algorithms and the communication capabilities of the network.

V.2.3 Requirement 3 : Server Recovery

To achieve successful failover after a group of components has stopped it is necessary to synchronize the activation of backup components. In a passive replication scheme this mainly involves to coordinate which backup replicas become active. This is hard since failover is done on a per component basis and each component possibly has several backup replicas. To ensure consistent system state after failover it has to be made sure that all backups that become active belong to the same failover group. Otherwise non-functional requirements might not be met since two components that were not intended to work together are accidentally activated simultaneously.

Application in the MCS scenario: When components of the primary chain fail and get deactivated all components in the backup chain need to become active and take over the role of the primary chain. In the presented scenario this is not likely, since there is only one backup replica per component, namely telecommand server B, telemetry server B, mission planning system B and archive B. However if a second backup chain C would be added things become more complex. The system could end up having some components fail over to replicas in chain C while others fail over to replicas in chain B. Components might be deployed in a way that leads to resource overuse in case of unintended failover orderings. The archive for example might need a reliable connection implemented as a real-time bus system, that is only available within nodes of one chain and not between the telemetry server node of chain B and the archive server node of chain C.

V.3 The CORFU Architecture

As motivated in the caste-study, component-based systems need fault-tolerance on a higher level of abstraction than single distributed objects. CORFU therefore introduces the concept of a *failover unit* (FOU) for component-based systems. The related modeling concept of a failover unit is described in the context of the MDDPro modeling tool [14]. We use this concept and transfer it to component middleware.

A failover unit contains a set of components that are interdependent on each other with respect to failure dependencies. This means that if one of the components fails, all components of the unit need to fail as well. It thus enables fail-stop behavior for a whole unit.

CORFUs FOU concept is based on passive replication. One unit is declared to be the *primary* unit. A primary unit actively processes requests and is made up of component instances that are all primary replicas within their component replication group. *Backup* FOUs are structurally identical to their corresponding primary FOU. This means that they consist of component instances that have the same interfaces and connection structure as their counterparts in the primary FOU. All component instances in the replica FOU are

backup replicas within their component replication groups. Backup units are not necessarily deployed on the same node constellation as the primary unit. It is possible to deploy all backup unit elements on one server node even if the primary FOU is deployed on a set of nodes.

Failover units can be described as a set of component instances that share a common task and role. Their role is either to be a primary or a backup unit. Since it is possible to have more than one backup for each FOU, each backup unit has a rank to determine the order of failovers (e.g. the primary will fail over to backup FOU number one. If FOU one fails afterwards, FOU two will become active and so on).

CORFU is implemented using the CORBA Component model as described in chapter III.2.2. CORFUs architecture consists of several aspects that are presented here in relation to the design challenges they address. Each challenge is presented together with CORFUs design decisions to overcome it. The challenges presented are (1) single component fault-tolerance, (2) integration into the deployment and configuration infrastructure and (3) failover ordering of replicas.

V.3.1 Challenge 1 - Single Component Fault-Tolerance

Problem: Providing passive replication for components requires means to (1) group components and treat them as replicas of one logical component instance, (2) a failover mechanism to activate a backup replica in the case of an error and (3) a fault detection mechanism that observes and reports when a system does not behave as expected.

In addition to this the nature of components add additional requirements for replication: (1) a component implementation can consist of several implementation artifacts that need to be replicated, (2) components also have connections to other components that need to be preserved during failover and (3) components are deployed in form of libraries that

are dynamically loaded into the process space of some generic container. Component initialization and fault-tolerance configuration therefore needs to be done through dedicated APIs.

Solution - Integrating FLARe through a fault-tolerant component server. FLARe provides object level fault-tolerance and can be adapted to support component-based passive replication. As described in section III.2.1 process level fault detection is done trough monitoring based on TCP/IP sockets. A failover mechanism based on ORB interceptors is provided to groups replicas based on so called RankLists containing an ordered list of object references. These references are associated with an object_id that identifies the logical object the replicas are part of.

In addition to FLAREs base functionality several adjustments have to be made to address the additional requirements of component replication. This includes

1. Enhancing the notion of a Replication Group to Components
2. Preservation of Component Connections
3. Providing a Fault-Tolerant Component Server

V.3.1.1 Enhancing the notion of a Replication Group to Components

As described in section III.2.1 a replication group in FLARE is realized through a rank list, that associates a group name with an ordered list of references. This list is cyclically distributed to all clients that then use it to contact backup replicas for failovers. This solution works on the level of single objects, since it uses object references within the rank list.

Components however often consist of several objects. One servant represents the component itself and each facet port and event sink are implemented by an additional servant. For components to be replicated it is therefore necessary to create associations between objects that form one component implementation.

The most seamless approach here is to register each of the components implementation objects at component start-up using a specific naming scheme for replication groups. The name of each object implementing a component aspect starts with the component instances name followed by the actual name of the port name the object represents. Let's assume the MCS archive component has a port named "data" for data retrieval and a port "mgnt" for administrative purposes. The replica group name of the main component object would be "Archive", and the port names for the two ports would be "Archive.data" and "Archive.mgnt". These names are used to register the objects with the component server POA that has a USER_ID id assignment policy and it is added to each interoperable object reference (IOR) of the objects through a server side portable interceptor. The same name is also used to register the replicas with the ReplicationManager and the StateSynchronizationAgent.

V.3.1.2 Preservation of Component Connections

Component connections within the CORBA Component Model are realized by storing object references to facet interfaces. These are registered with the context of each component that uses the facet. A similar mechanism is used for event ports.

FLARe already provides functionality to distribute failover RankLists to clients that then can perform failover functions. Since all facet objects are included in this list, connections are automatically kept valid: When a component tries to use a receptacle connected to a facet object that is no longer accessible, it will automatically fail over to an appropriate replica.

V.3.1.3 Providing a Fault-Tolerant Component Server

A component server is a generic process in the DAnCE infrastructure that hosts component instances. The library containing the component implementation is loaded into the process space of a component server by a DAnCE NodeApplication instance. A Container

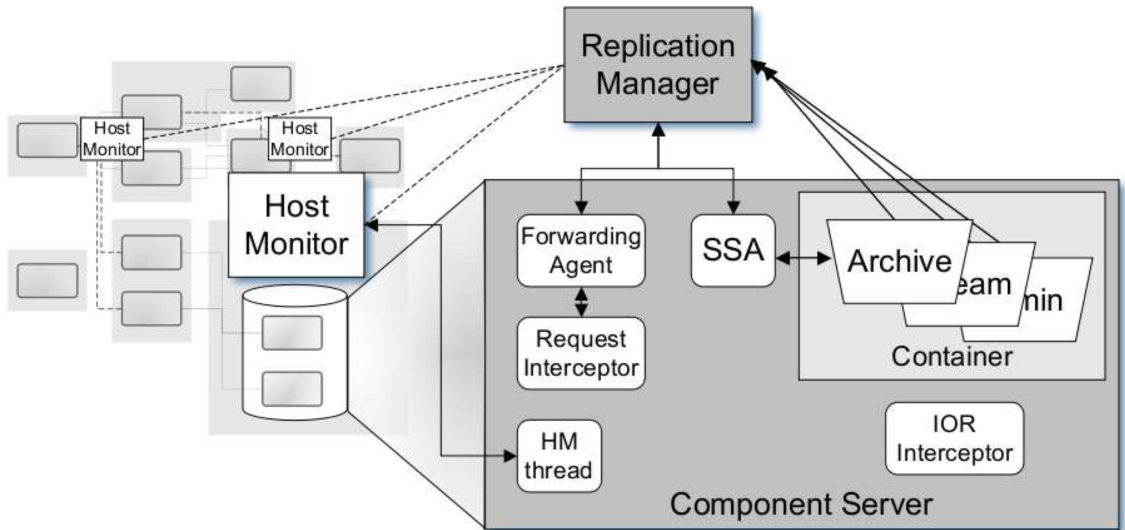

Figure V.2: Structural Overview of a Fault-Tolerant Component Server

is the run-time infrastructure within the component server that provides a component with APIs to interact with this run-time. Based on XML meta-data the container will instantiate, configure and start components.

To allow support for component-based fault-tolerance in CORFU, a fault-tolerant component server has been developed. It includes all common functionality of a component server and contains additional fault-tolerance functionality. Figure V.2 gives a structural overview of the adjustments have been made to host fault-tolerant component replicas. These can be summarized as follows:

1. Per-process initialization tasks, such as registration with the HostMonitor, initialization of the StateSynchronizationAgent and the ForwardingAgent and registration of those with the ReplicationManager.

2. A fault-tolerant session container is instantiated that allows per component registration functionality such as POA registration, using the replica group id, embedding of this id into each replica object reference and registering the component with the ReplicationManager.

V.3.2 Challenge 2 - Integration into the Deployment and Configuration Infrastructure

Problem: The interfaces of the OMG Deployment and Configuration (D&C) Specification as described in section III.2.3 are not providing fault-tolerance functionality. However provisioning of fault-tolerance on groups of components requires to integrate into the system model of the D&C infrastructure. CORFU needs to be standard compliant and yet minimize performance overhead at run-time. This challenge includes the mapping of the component deployment hierarchy to the FLARe system model hierarchy. While the D&C model consists of *nodes* and *components*, FLARe uses a model that contains *objects* residing in *processes* that run on *hosts*. The second part of the challenge is integrate failover unit related fault-tolerance properties into standard D&C deployment plans that do not have any notion of replication.

The following partial solutions address this challenge:

1. Deployment Plan Preparation
2. Design of a FaultCorrelationManager
3. Mapping for FLARe and D&C System Models

V.3.2.1 Deployment Plan Preparation

For CORFU to be standard conform failover units need to be expressed through the means available in the deployment plan specification of the OMG. A deployment plan is static in nature. All component instances will be started together and are expected to operate throughout the active phase of a deployment. Adding or removing particular component instances during system lifetime is not supported by standard D&C interfaces. Since Failover units need to be shut down prior to the shutdown of the whole system, CORFU requires to split deployment plans into several sub-deployment plans based on failover information.

CORFU provides the algorithm SPLIT-FOU (algorithm 1) that performs such deployment plan splits. The algorithm has to fulfill the following post conditions to achieve a correct split.

1. Each failover unit needs to be represented by a separate deployment plan.
2. All component instances of the original plan must be contained within one of the sub deployment plans.
3. Connections between component instances residing in different sub plans need to be maintained by inserting external references and creating new connections among them.

The algorithm has two input data structures: A deployment plan containing all component instances and connections within a system and a failover unit specification that associates instances in the deployment plan with failover units. By separating the declaration of the failover units from the plan, these two aspects are made orthogonal. This allows to define different fault-tolerance scenarios for one deployment plan and without modifying the plan itself.

A deployment plan D is defined as $< n, u, r, I, C >$, where n is the string id of the plan, u is the name of the failover unit, this plan represents, r is the rank for the failover order of this plan, I is a list of all component instances in the deployment and C is a list of all connections between two components within the deployment. Each instance I_i is defined as $< n, m >$, where n is the name of the instance and m the name of the node an instance is deployed on. Each connection C_i is an ordered pair of endpoints. Endpoints exist in two forms: External endpoints e_i refer to a connection outside of the current deployment through a stringified path of the form `<deployment id>/<instance id>/<port id>`. The other form are internal endpoints p_i that refer to a component instance within the current deployment through a path of the form `<instance id>/<port id>`. The first element in the pair represents a component port that *uses* a service from another component,

while the second entry *provides* the service. The failover unit declaration F is defined as $< n,u,r,J >$ where n is the concrete name of the failover unit, u is the group name of failover units that are replicas of each other, r is the rank of the unit within its group and J is the set of instance names from the deployment plan that are contained in this failover unit.

The algorithm operates in two phases. The first phase will create a new deployment plan for each failover unit and will will populate them with the correct instances. All components that are not members of any failover unit are copied into an additional deployment plan. In the second phase each connection is analysed. If both instances are still in the same plan, it is simply copied into this sub-plan. Otherwise a connection to an external endpoint referencing the correct component port is added to each of the two sub-plans that contain these components now.

Algorithm 1 has a relatively high time complexity of $\mathcal{O}(n^2 log(n)) + \mathcal{O}(m * n)$, where n is the number of instances per plan and m the number of connections per plan. Since this is an offline algorithm that is run before system deployment it does not need to be especially optimized.

V.3.2.2 Design of a FaultCorrelationManager

CORFU introduces the FaultCorrelationManager (FCM) to manage fault-tolerance functionality for failover units.

To integrate the FCM into the existing D&C infrastructure, the Decorator Pattern [6, p.175] is applied. As shown in figure V.3 the FaultCorrelationManager implements the ExecutionManager interface and can therefore be accessed by any service that uses the ExecutionManager interface. The PlanLauncher that is responsible for passing deployment plans to the ExecutionManager can now use this functionality.

In the context of the Decorator pattern, the FCM plays the role of a ConcreteDecorator and the ExecutionManager is a ConcreteComponent. The Decorator Role in this case is

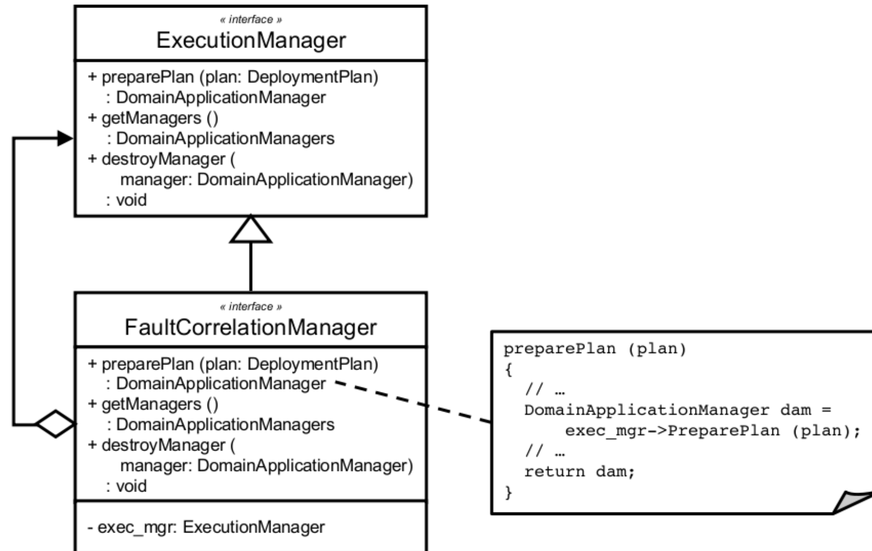

Figure V.3: Application of the Decorator Pattern for the FaultCorrelationManager design

not implemented as a class, but represented through the CORBA run-time that allows to access the manager interface through an IOR. For the client it is transparent whether this reference is pointing to a standard ExecutionManager or to a FaultCorrelationManager.

The FCM enhances the three methods of the interface with additional functionality. Within these method implementations the calls are forwarded to the ExecutionManager.

The main tasks are performed at component deployment through the `preparePlan ()` method. This includes creating an internal representation of the complete system as described in section V.3.2.3 and functionality to order per-component replica groups as described in section V.3.3.

The algorithm $SPLIT-FOU$ is not part of the functionality of the FCM. This is due to the fact that the ExecutionManager interface allows access to each DomainApplication-Manager and the user therefore needs to have complete knowledge about which deployment plans are running. Creating deployment plans automatically would break this transparency. However, it is envisioned to implement SPLIT-FOU in the context of a domain specific modeling tool.

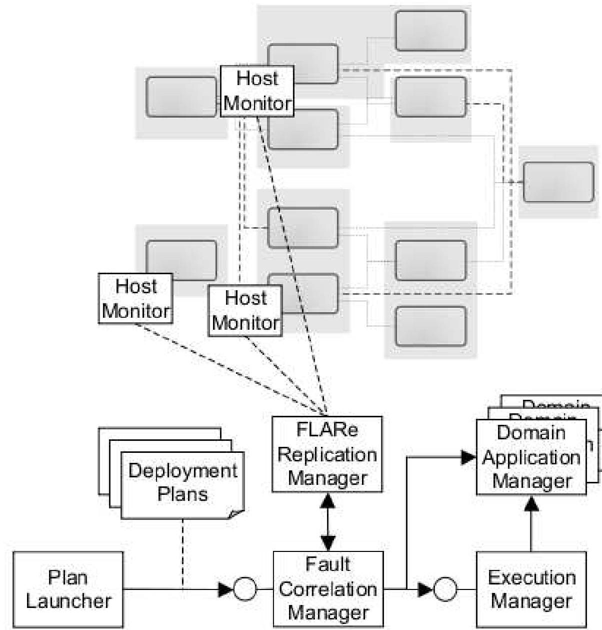

Figure V.4: Structure of a Fault-Tolerant Component System

V.3.2.3 Mapping for FLARe and D&C System Models

In figure V.4 the structure of a deployed system is summarized. The ReplicationManager receives information about process failures from HostMonitors that are deployed on each server. A HostMonitor observes all fault-tolerant processes of a system and reports the process id to the ReplicationManager if a process crashes. The Replication system model therefore contains hosts and processes running on these hosts. It also associates single objects with the process they are running on, which results in a three layer hierarchy including hosts, processes and objects. A deployment plan has a two layer system model, with nodes and components that are deployed on those nodes.

The ReplicationManager needs to report failures to the FCM. The FCM then determines which other components to stop based on the failure. To achieve this, a mapping between both hierarchies is needed so that the FCM can process failure information sent by the ReplicationManager in the most efficient way.

The proposed mapping is structured as follows: As a basic design decision we chose

to identify node names with host names. Every server in the system that is observed by a HostMonitor will also represent exactly one node named after the network name of the server machine. We furthermore annotate components in the deployment plan with a property that identifies the replication group of component. This name is then passed to the ReplicationManager on registration as object_id. Since an object_id represents a group of replicated objects, it is still not guaranteed that a component is uniquely identified by tuple of host name and object_id. Since it is not advisable to host several backup replicas of the same object_id on the same physical host due to risk of loosing both replicas through a crash of this server, it is appropriate to restrict the number of replicas per host and object_id to one. This results in unique identification of components within one host.

Based on these decisions the following callback interface is implemented by the FCM and registered with the ReplicationManager to receive notifications about process crashes.

```
typedef sequence<string> ApplicationList;

interface FaultNotification {
  void app_failure (in string host,
                    in ApplicationList applications);
};
```

The ReplicationManager passes the host name of the crashed process as host parameter and the FCM interprets it as node name. The applications parameter contains a list of object_ids that were hosted in the crashed process. Note that the FCM does not need to know in which processes a component is hosted in.

During deployment the FCM analyses the deployment plans to populate the following data structures that help it to react on failure notifications:

1. A hash map I associates component *instance* names as keys with the id of the deployment plan they are hosted in.

2. For each node a map O is maintained that uses the object_id as a key to find the component instance name that is a local replica for this object_id on the node. These

node maps themselves are stored within a hash map N that allows to find them by using the node name as a key.

3. Each created DomainApplicationManager is stored in a map M with its deployment plan id as key.

Based on these data structures reaction time on failure notifications is optimized since all access times to these maps are small.

The algorithm 2 operates on these maps to process fault notifications during system run-time. The processing is done in two phases. In phase one, all affected failover units, represented as deployment plans are determined based on the failure information. This phase uses the internal maps. The second phase uses exiting D&C infrastructure, namely the DomainApplicationManagers to stop all component applications that belong to these deployment plans.

V.3.3 Challenge 3 - Failover Ordering of Replicas

Problem: FLARe uses passive replication on a per-component basis. This means that a backup replica takes over processing when the primary replica of a component fails. For this purpose the ReplicationManager maintains a so called RankList for each fault-tolerant object. The references within this list are sorted in the order in which they will become active, starting with the first backup replica. Since the ReplicationManager has no understanding of component groups it is hard to coordinate failovers across several individual components. It has to be guaranteed that the failure of a primary failover leads to the activation of all backup replicas in the next backup failover unit. It needs to be prevented that component replicas from more than one backup failover unit are active at the same time.

Solution - Failover Constraints. Our solution approach is to modify the Replication-Managers algorithm such that it can process constraints. The per object order determined

```
typedef sequence<string> HostList;

struct RankListConstraint
{
  string object_id;
  HostList hosts;
};

typedef sequence<RankListConstraint> RankListConstraints;
```

Figure V.5: IDL Declaration of RankList Constraints

by such constraints needs to be maintained. As shown in figure V.5, constraints are defined as sequences of host-names associated with a replica object id. The first host list entry indicates, where the primary is hosted and the following hosts contain backup component replicas. Since every host only has one replica of the same group the constraint contains enough information for the ReplicationManager to uniquely identify a replica.

The FaultCorrelationManager creates constraints based on information from the deployment plan. Each deployment plan, representing a failover unit needs to be assigned a rank within its group of failover unit replicas. The FOU-ORDERING algorithm for failover unit based replica ordering is described in algorithm 3. All components within a unit will be assigned the units rank. The constraints are updated using this algorithm whenever the deployment changes. This happens if new deployment plans are loaded or when failures occur and deployments are removed.

Algorithm 1 SPLIT-FOU (D)

```
Data: A deployment plan D
Data: A failover unit definition F
Output: A set of deployment plans S

for each F_i ∈ F do
    create new deployment plan s ∈ S;
    set u of s to u_i;
    set name n of s to name(D)+name(F_i);
    set rank r of s to r_i of F_i;
    for each J_k ∈ F_i do
        find I_l ∈ I | name(I_l) = J_k;
        copy I_l to s;
        mark I_l as processed;
    end
end
create new deployment plan s ∈ S;
for each unmarked component instance I_i ∈ D do
    copy I_i to s;
end
for each C_i ∈ D do
    p_1 = first endpoint of C_i;
    i_1 = instance_id(p_1);
    find plan s_1 ∈ S | i_1 ∈ s_1;
    n_1 = name(s_1);
    p_2 = second endpoint of C_i;
    i_2 = instance_id(p_2);
    find plan s_2 ∈ S | i_2 ∈ s_2;
    n_2 = name(s_2);
    if n_1 = n_2 then
        copy C_i to s_1;
    else
        create external endpoint e_1 with path n_2 + p_2;
        add connection < p_1, e_1 > to s_1;
        create external endpoint e_2 with path n_1 + p_1;
        add connection < e_2, p_2 > to s_2;
    end
end
```

Algorithm 2 FAILURE-REACTION (h,F)

```
Input: host name h
Input: list of failed object ids F
Data: Component Instance Map I
Data: Node Map N
Data: DomainApplicationManager Map M

look up object_id map O with key h in N;
create empty set P of deployment plan names;
for each F_i ∈ F do
    look up instance name i with key F_i in O;
    look up plan name p with key i in I;
    if p is not in P then
        add p to P;
    end
end
for each p ∈ P do
    look up DomainApplicatonManager m with key p in M;
    retrieve list of ApplicationManagers A through m.getApplications ();
    for each NodeApplication a ∈ A do
        call m.destroyApplication (a);
    end
end
```

Algorithm 3 FOU-ORDERING

```
Data: List of deployment plans D
Output: A constraint list L
partially sort plans in D by their ranks;
for each plan d ∈ D do
    for each instance i ∈ d do
        get object_id o property from i;
        get node name n property from i;
        append n to list entry of L with object_id o;
    end
end
```

CHAPTER VI

RESULTS

CORFU provides advanced fault-tolerance capabilities for DRE systems. We evaluate this claim using two different approaches. First, we do a conceptual analysis of the development effort by comparing object-based development of fault-tolerant applications with development using the CORFU infrastructure. Second, we present measurements of CORFUs' timing behavior. These include measurements of client-side failover latency and of the round-trip latency of failover unit fail-stop events.

VI.1 Benefits of Component-based Fault-Tolerance compared to Object Level Fault-Tolerance

Developing applications that support distributed object-oriented fault-tolerance as provided by FLARe involves additional effort with respect to application development. This evaluation qualifies those efforts and contrasts them with the component based fault-tolerance approach CORFU provides.

Development obligations of object-oriented fault-tolerance: FLARe requires different means to implement fault-tolerance on the server side, where the object to be replicated resides, and on the client side, which uses replicated services. We will therefore separately consider the obligations for server applications and client applications. We furthermore distinguish between (1) object implementation obligations that each CORBA servant needs to implement to integrate into the fault-tolerance infrastructure, (2) initialization obligations an application needs to perform to use FLARe functionality and (3) configuration obligations at start-up that configure fault-tolerant aspects of the application.

Figure VI.1 gives an overview of all obligations related to server side development. Each object implementation needs to provide callback interfaces to allow CHESS to do

CORBA 2.x Server Obligations		
Object Implementation	Initialization	Configuration
1. Implementation of get_state/set_state methods 2. Triggering state synchronization through state_changed calls 3. Getter & setter methods for object id & state synchronization agent attributes	1. Registration of IORInterceptor 2. HostMonitor thread instantiation 3. Registration of thread with HostMonitor 4. StateSynchronizationAgent instantiation 5. Registration of State Synchronization Agent with Replication Manager 6. Registration with State Synchronization Agent for each object 7. Registration with Replication Manager for each object	1. ReplicationManager reference 2. HostMonitor reference 3. Replication object id 4. Replica role (Primary/Backup)

Figure VI.1: Development Obligations for Server-Side Fault-Tolerance

state synchronization. State synchronization additionally requires notification of the StateSynchronizationAgent about state changes as discussed in chapter IV. Getter and setter methods have to be provided to give access to the supported replication object name and needed agent references.

The main effort on the server side apart from object implementation is related to initialization of the FLARe infrastructure. The server application developer needs to programmatically perform the following tasks: An IOR interceptor has to be instantiated to allow the annotation of exposed server object references with object id information. For the HostMonitor to observe a server application, a local thread has to be initialized. The application then has to be registered with the monitor. The same procedure is necessary for a process wide state synchronization agent. The agent needs to be instantiated and registered with the ReplicationManager to receive information about other present object replicas. In addition to these process wide initialization steps each object exposed by a server application needs to be registered with the state synchronization agent and the ReplicationManager.

While the previously described steps need to be done programmatically, some aspects need to be configured at application start-up time. This includes passing of the references to

CORBA 2.x Client Obligations	
Initialization	Configuration
1. Registration of Client Request Interceptor 2. ForwardingAgent instantiation 3. Registration of ForwardingAgent with ReplicationManager	1. ReplicationManager reference

Figure VI.2: Development Obligations for Client-Side Fault-Tolerance

the ReplicationManager and the HostMonitor as well as the configuration of fault-tolerance properties of the server objects, such as their role and the id of the logical object they represent. This is either done through command line parameters or a proprietary file format that is read by the application.

Figure VI.2 summarizes the obligations on the client side. Since FLARes architecture provides a failover mechanism that is transparent to the client application as possible this does not involve as many steps as server implementation. However there are still several obligations that need to be performed correctly.

A client request interceptor needs to be initialized in order to transparently detect failures and forward requests to backup replicas. To inform the request interceptor about available failover targets, a ForwardingAgent needs to be set up and registered with the ReplicationManager. At start-up a client then must be configured with the object reference of the ReplicationManager via command-line parameters or other means.

Consequences for application development: All the necessary obligations presented here result in considerable accidental complexities in application and system development. Being required to manually implement all initialization steps in clients and servers increases the risk of accidentally omitting or confusing steps. This is even more problematic since debugging of fault-tolerance aspects is hard due to its distributed nature.

In addition to these threats for quality and correctness, this approach also limits the flexibility of system implementation. The number and type of object replicas per server

process are determined by manually written code. This means that any change in positioning of replicas in the system is limited. Either each object needs to be reside in a separate application to allow flexible positioning on replicas on according hosts. This however incurs additional resource consumption related to FLARes' infrastructure. The alternative is to collocate several objects which requires the adaption of application code. This increases the time needed to adapt a system to new requirements and thus complicates system evolution.

Benefits of CORFUs' component-based approach: By integrating FLARe functionality into a fault-tolerant component server, CORFU overcomes many of these limitations of traditional object-oriented fault-tolerance approaches. Server and client side capabilities are available within the same component server. Since CORBA objects often play both roles of server and client at the same time this is a suitable architectural decision. We present the benefits of the component server approach by relating them to the three different types of obligations as presented earlier.

1. **Object Implementation:** Component executors, being the concrete implementation artifact of a component interfaces technically are CORBA servants. They therefore have to fulfill the same obligations as in the object-oriented case. However CCM provides code generation functionality in the form of the IDL and CIDL compilers that automatically can create necessary code artifacts.

2. **Initialization:** Most of the steps of client and server initialization can be done automatically. Instantiation of the state synchronization agent, the ForwardingAgent and the HostMonitor thread are not related to hosted objects. The fault-tolerant component server, therefore, hides the complexity of initializing these entities from the component developer. The registration of individual components with the framework also can be done automatically by a fault-tolerance aware session container. The necessary information, such as the role and object id of a component can be submitted using configuration attributes provided in the deployment plan specifications.

3. **Configuration:** Instead of using proprietary mechanisms on a per-application level the component server approach enables the use of standardized configuration mechanism provided by the D&C specification. Special fault-tolerant component attributes can be defined and initialized within the deployment plan through so called configuration properties. This still leaves the obligation to configure these properties. But instead of doing this in scripts in the form of command line parameters or using proprietary solutions that might even vary from application to application, this standardized approach uses existing infrastructure to actually instantiate the system.

Conclusion: CORFU increases the transparency of using fault-tolerance mechanisms for server development as well as for client development. This allows for the application developer to focus on business logic implementation while fault tolerance aspects can be added and configured orthogonally. It is possible to collocate fault-tolerant components without changing their implementation code. CORFU therefore also substantially improves the flexibility of system deployment and system evolution. In addition to that there are fewer possibilities of accidental faults in application development, since the initialization is done in a well tested and stable way by the component server.

VI.2 Experimental Results

This section presents experiments that evaluate the timing behavior of CORFU. These experiments allow a better understanding of latencies involved in the failover mechanisms and clarifies for which timing requirements CORFU is sufficient. The first experiment evaluates failover latency as experienced by a client application. The second one focuses on timing latency of the coordinated shutdown of a failover unit.

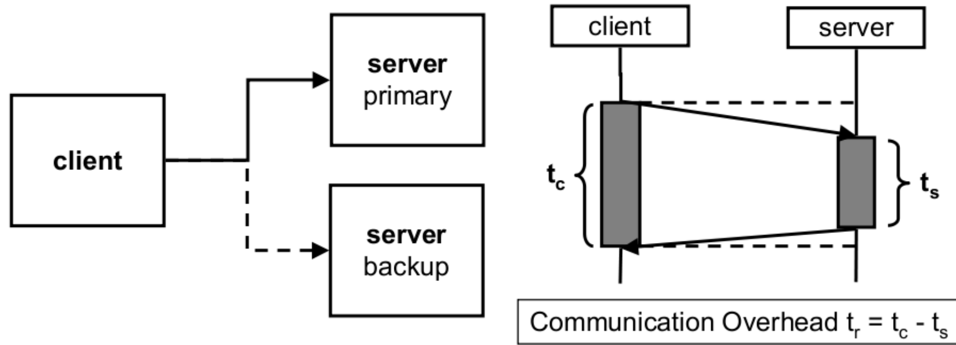

Figure VI.3: Experiment Setup for Failover Latency Measurement

VI.2.1 Testbed

All experiments have been conducted on ISISLab[1], a LAN virtualization environment with identical blades connected through 4 Gbps switches that allow for dedicated links per experiment. The blades each have two 2.8GHz Xeon CPUs and 1 gigabyte RAM. The Fedora Core 6 Linux distribution rt11 real-time kernel patches is used as operating system. The enhancements to FLARe and the CORFU implementation are based on TAO version 1.6.8 a real-time CORBA implementation and CIAO version 0.6.8, which is an implementation of the CORBA component model. CORFU and all testing applications have been built using the GNU compiler collection gcc version 3.4.6.

VI.2.2 Failover Latency

Experiment Setup: This experiment compares the failover latency a client experiences for CORBA 2.x applications and component-based applications.

Figure VI.3 shows the basic setup of the experiments. A client application periodically calls a server application that is replicated. The period is 200 milliseconds and the execution time of each task is 20 milliseconds. We used a CPU worker component of the system execution modeling tool CUTS[2] that allows to simulate a defined processing time in millisecond accuracy. With each call the server sends back the actual time from the beginning

[1] http://www.isislab.vanderbilt.edu

of processing the request to the end of processing. This can be more than the 20 milliseconds since the process might be preempted by other processes on the same host. The client also measures the time from issuing the request on the server until it receives a response. By subtracting the server side processing time from the measured response time, the time for communication can be calculated.

After a defined number of calls the server will simulate a fault by shutting down. This causes the client to fail over to the backup replica of the server. At this moment the response time in the client is expected to increase due to the fact that the connection error has to be detected and a new connection to the backup replica is established.

All primary servers are hosted on one host, the backup servers are hosted on a separate host. The clients also a deployed together on a dedicated host and all CORFU infrastructure entities, such as the ReplicationManager and the D&C run-time are hosted separately to not interfere with the timing measurements.

This setup has been implemented in two variants. Variant 1 is object-oriented and consists of a client and a server executable that directly use FLARe functionality. Variant two is component-based and uses CORFUs' fault-tolerant component server. Each variant is used in three different experiment configurations. Configuration one runs one group of client, primary replica and backup replace, configuration two runs two such groups in parallel and configuration three has four applications that operate at the same time. Each measurement configuration is repeated 100 times and the average is used for the evaluation.

Measurement Results: An example for a single measurement for failover latency is given in figure VI.4 (1), which represents the component-based case with one application set running. The ten invocations before and after a failure event are recorded. The first 10 invocations show a communication overhead between zero and one milliseconds, which represents failure free communication with the primary server. On the clients attempt to contact the server at invocation eleven the failover occurs since the server shut down after ten invocations. In this case the latency increases to four milliseconds due to the processing

time within the ORB to detect the CORBA exception indicating the servers unresponsiveness and the forwarding of the request to the backup replica.

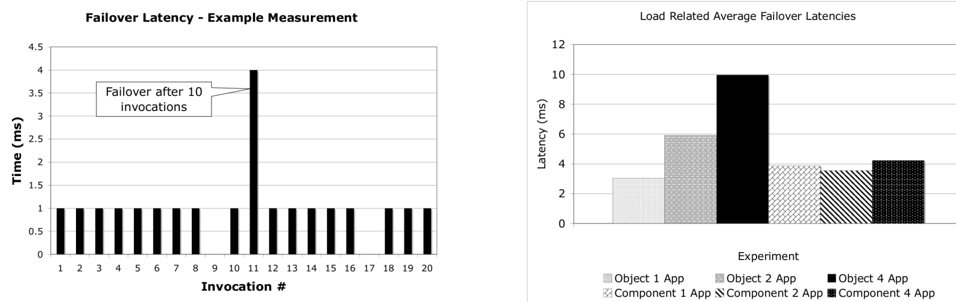

Figure VI.4: Failover Latency Measurements

Figure VI.4 (2) shows the average latencies as measured in all six configurations. The three CORBA 2.x based object-oriented experiment with one application shows a communication overhead of three three milliseconds, while the corresponding component-based experiment has a latency of four milliseconds. This shows that the extra cost for the component-based fault-tolerance with 25 percent additional overhead is relatively small. Looking at the configurations with two and four applications we can see that the component-based experiments stay constant around four milliseconds of latency, while the object-oriented examples have growing response times. This is not directly related to the failover-mechanism but reflects the implicit differences between the experiment variants, since the executables start processing right away while a component is first loaded into the container and then triggered later on to start processing. Nevertheless the results show that there is no unreasonably high overhead for component based fault-tolerance.

VI.2.3 Failover Unit Shutdown Latency

Experiment Setup: The second experiment is designed to give insight into the latency

involved in the process of shutting down a failover unit. This latency is due to several factors:

1. Error detection and notification delay from the failure of a component to the start of processing its notification by the FCM
2. Reaction delay within the FCM to determine which components are affected and which deployments therefore need to be shut down
3. Shutdown time using the D&C services, namely the DomainApplicationManager and it's node application interfaces to destroy the affected node applications.

The structure of the experiment and its logical sequence of events is shown in figure VI.5. The setup includes six processing nodes of which one node is dedicated for the CORFU management entities, such as the ReplicationManager, the FCM, the ExecutionManager and other elements of the D&C run-time. The other five nodes have a HostMonitor deployed to observer the system state per node. Each node hosts one component for each of five deployed failover unit. There is one primary failover unit that includes one component per node, named A_0 to E_0. This failover unit is replicated four times through the backup failover units one through four. Each of the backup units contains replica components A_n to E_n of each component in the primary unit. The failover order of the units corresponds to their number. The experiment will inject failures in the currently active component, leading to a failover sequence of primary FOU, backup FOU 1, backup FOU 2, backup FOU 3 and finally backup FOU 4. Each experiment run therefore allows us to measure four failover latencies.

Due to the need for consistent time, all measurements are taken on node-1 in the ReplicationManager and the FaultCorrelationManager. This prevents the need for synchronized clocks. The measurements are done in the following sequence:

1. A failure is provoked in component A_n of the active FOU.

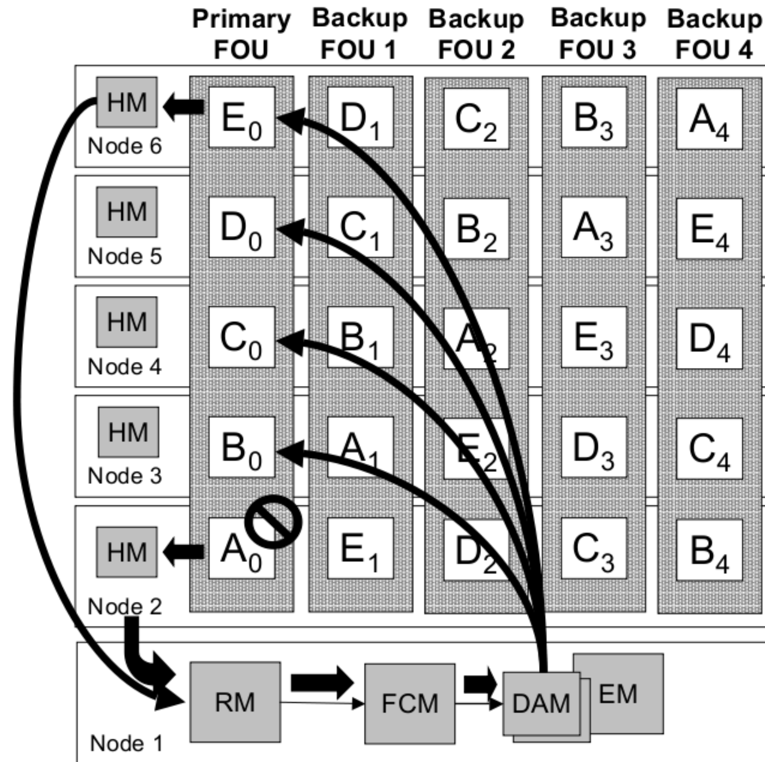

Figure VI.5: Experiment Setup for Failover Unit Shutdown Latency Measurement

2. The failure is detected by the HostMonitor and reported to the ReplicationManager.

3. The ReplicationManager takes a time-stamp at time t_1 when it receives the failure notification and notifies the FCM about the occurred failure. The FCM takes a times tamp at time t_2 when it is notified about a failure.

4. The FCM does look-up operations in its internal maps to determine which FOU deployment plans need to be shut down and takes a time-stamp t_3 after finishing this look-up.

5. The FCM will then access the DomainApplicationManager to retrieve all node applications for the corresponding deployment plans and then will iterate through them to shut them down. After the last call is returning, a time-stamp at t_4 is taken to indicate the finishing of the shutdown request.

6. The ReplicationManager will be notified about all the shutdowns of the affected components by the HostMonitors. On reception of the last shutdown notification, a timestamp for t_5 is taken that represents the time when the FOU is completely shut down and a client would failover to a backup replica no matter which component in the FOU it tries to access.

Measurement Results: The measured times allow us to determine the following latency times:

$$\textit{Round-trip Time } t_{\text{round-trip}} = t_5 - t_1 \tag{VI.1}$$

The round-trip time is the sum of all latencies involved in the shutdown of a failover unit. This includes failure detection, reaction time within the FCM and shutdown time by the D&C run-time.

$$\textit{Reaction Time } t_{\text{reaction}} = t_3 - t_2 \tag{VI.2}$$

The reaction time is the time spent within the FCM between the failure notification and the start of the shutdown process. This basically is the time needed to perform the FAILURE-REACTION algorithm 2 and to serialize incoming notifications into a thread-safe queue to ensure correct processing of parallely detected errors.

$$\textit{Shutdown Time } t_{\text{shutdown}} = t_4 - t_3 \tag{VI.3}$$

The shutdown time as measured by the FCM allows us to get an understanding which proportion of $t_{\text{round-trip}}$ is not related to the D&C shutdown mechanism which cannot be changed without breaking the standard.

Figure VI.6 shows minimum, maximum and average round-trip and shutdown latencies for fail-stop measurements. Reaction latencies have not been displayed since they are negligible compared to the other types of latency.

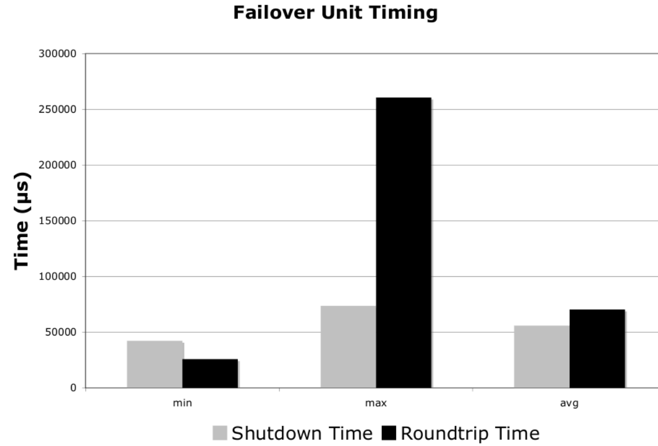

Figure VI.6: Measurement results for fail-stop latencies

As we can see from the graph, CORFU has the following latency characteristics:

$$\textit{Average Latency Time } \bar{t}_{\text{round-trip}} = 70.59ms \tag{VI.4}$$

of which the shutdown time represents a huge proportion although it does not fluctuate as much.

$$\textit{Average Shutdown Time } \bar{t}_{\text{shutdown}} = 56.04ms \tag{VI.5}$$

For the internal reaction time of the FCM, experiments show that is no crucial factor in timing behavior:

$$\textit{Average Reaction Time } \bar{t}_{\text{reaction}} = 0.24ms \tag{VI.6}$$

VI.2.4 Discussion

Based on the experiments we performed several characteristics of CORFU are exposed. Using a client-side failover mechanism allows for short failover latencies, since communication with the central replication manager in the instant of a failure is avoided. This would be a bottleneck in performance of large-scale systems. As shown by the first experiment, this client side failover latency is relatively small, being three milliseconds for the object

variant. Having evaluated the benefits for CORFU concerning application development and system deployment we also needed to ensure that this does not drastically degrade performance and therefore render the solution unusable for DRE applications. As our experiment shows, client failover in CORFU is comparable in performance and occurs only minimal overhead, having an average response time of four milliseconds.

Compared to the client failover latency the failover unit shutdown latency with 70 milliseconds in average is relatively high. The reason for this is partly to be found in the iterative way a deployment has to be shutdown based on the domain application and node application interfaces. Another source of long response times is the communication time between the different entities, such as the HostMonitors, the ReplicationManager and the FCM. The internal reaction time of the FCM to determine deployments that are affected by faults is already optimized through the use of hash maps with close to constant access times. With an average beneath 0.25 milliseconds it does not substantially contribute to the overall processing time.

Based on these sources of overhead, we envision three approaches to reduce the round-trip latency for failover unit shutdown:

Parallelized Shutdown To reduce the shutdown latency the calls initializing shutdowns for affected node applications can be parallelized instead of being done in sequential order. A suitable mechanism to do so is Asynchronous Method Invocation as defined in the CORBA standard. This allows the FCM as a client to issue all shutdown requests without having to wait for their response in between. This would lead to significant reduction of the shutdown time, especially in large deployments.

Collocation of Management Entities Some communication paths, especially between ReplicationManager, FCM and ExecutionManager can be optimized by collocating these entities into the same process space. This greatly reduces communication times since the network stack can be avoided and in process communication mechanisms are used instead.

RTCORBA For the communication paths that need to go through the network, communication can be made more reliable and deterministic by using RTCORBA features, such as the real-time scheduling service, private connections, pre-allocation of connections and end-to-end priorities.

Although there still is potential for performance improvement, the measurements show that CORFU is suitable for DRE systems and offers comparable performance to distributed object-oriented fault-tolerance.

CHAPTER VII

CONCLUDING REMARKS

Research on fault-tolerant DRE systems often has focused on solutions that simply focus on the fault tolerance related aspects of a system. Application development effort and system evolution often is not taken into account. Existing frameworks mostly use an object-oriented paradigm to provide fault-tolerance.

Our work shows that applying the component-based development paradigm can improve transparency of fault tolerance aspects in the application development process and therefore fosters more flexible system structures and better support for system evolution. We showed how this approach also increases the speed and quality of application development. Through measurements of the CORFU infrastructure we showed that component-based fault-tolerance can be provided within required performance limits.

VII.1 Lessons Learned

Through our work on component-based fault-tolerance for distributed real-time and embedded systems we gained a better understanding of the domain, which is summarized in the following lessons learned:

1. Fault Tolerance affects all aspects of a system and introduces a new dimension of complexity. It is therefore hard to capture all fault tolerance aspects in a comprehensive middleware framework. Application characteristics differ greatly even within the DRE domain, which affects used protocols, architectural concepts applied and technologies chosen. Each of these choices might require different approaches to fault-tolerance.

2. Development of fault tolerant systems can benefit greatly from integration into middleware. Although there is no one-size-fits-all solution central fault-tolerance aspects can be captured in frameworks through intelligent design approaches. CHESS is an example that uses design patterns to separate application specific concerns from common fault-tolerance mechanisms and thus increases the level of automation of replica state synchronization.

3. Component-based Middleware allows for greater fault-tolerance transparency. As demonstrated by CORFUs' fault-tolerant component server, the component-based development paradigm and lightweight fault-tolerance integrate very well, allowing the hiding of much complexity that exists in this domain.

4. Layering and separation of concerns fosters flexible and architectures. This becomes clear in the design of the fault correlation manager. By building the failover units on top of the existing object based approach and separating concerns through failover constraints, the fault correlation manager design and implementation could be kept small and focused on its main task to analyze the system infrastructure and react on failures using other existing software, namely the deployment and configuration infrastructure.

5. Performance of Fault Tolerance is hard to measure due to singular nature of failures, non-determinism in network, operating system and middleware. Since faults are no periodical events in systems expecting fail-stop behavior the setup of experiments is complex. Each measurement can only measure a very limited number of faults before the complete system has to be restarted. Additionally the nature of distributed systems makes it hard to gather reliable timing information due to network jitter, operating system scheduling and other sources of non-determinism. Experiments and testing scripts need to be highly automated to allow a sufficient number of single measurements.

VII.2 Future Work

The work on CORFU touched on many aspects of fault-tolerance in the context of component based system. Future work in this area needs to be done to achieve a comprehensive solution that integrates all aspects of fault-tolerance and component-based software development.

- **Optimization of Group Failover in the Fault Correlation Manager and DAnCE:** As discussed in the experimental results section, failover latency of failover units needs to be optimized.

- **Transactions within and across Failover Units for advanced state consistency guarantees:** So far, CHESS provides consistency mechanisms on the level of single components. Certain applications require consistency guarantees for groups of components, such as failover units. Transactional semantics need to be introduced to enforce stronger guarantees of consistency.

- **Extending the Fault Model to Network Failures:** DRE systems might need to operate in environments with highly unreliable communication channels. In such scenarios fault models need to include network failures as well as host and process failures. Further research has to be done to enhance FLARe and CORFU to deal with partitioning in networks and reconciliation of state and failover information after reestablishment of connectivity.

- **Integration of CORFU with Domain Specific Modeling tools, such as MDDPro [14]:** CORFU provides a run-time solution for groups of components with fault dependencies. To integrate these concepts into a domain specific modeling language allows for a comprehensive engineering approach to fault-tolerance. The SPLIT-FOU algorithm in particular integrates well into a modeling environment and eases the burden of the system deployment planner.

- **Additional Fault Tolerance Aspects for Component Deployment Infrastructures:** To avoid single points of failure, all entities in a component based run-time system need to be fault-tolerant. Further research needs to be done to apply object-based fault-tolerance to entities such as the execution manager, the node managers and other D&C entities as well as to the fault correlation manager itself.

REFERENCES

[1] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report 01145, LAAS-CNRS, Toulouse, France, 2001.

[2] J. Hill J.M. Slaby S. Baker and D.C. Schmidt. Applying system execution modeling tools to enterprise distributed real-time and embedded system qos. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[3] Jaiganesh Balasubramanian, Sumant Tambe, Aniruddha Gokhale, Chenyang Lu, Christopher Gill, and Douglas C. Schmidt. FLARe: a Fault-tolerant Lightweight Adaptive Real-time Middleware for Distributed Real-time and Embedded Systems. Technical Report ISIS-07-812, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN, May 2007.

[4] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. Jagr: an autonomous self-recovering application server. *Autonomic Computing Workshop, 2003*, pages 168–177, June 2003.

[5] George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 132, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1972-5.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[7] Boris Gruschke. A new approach for event correlation based on dependency graphs. In *In 5th Workshop of the OpenView University Association*, 1998.

[8] John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. *Software Engineering, International Conference on*, 0:160, 2003. ISSN 0270-5257.

[9] *Deployment and Configuration of Component-based Distributed Applications, v4.0*. OMG, Document formal/2006-04-02 edition, April 2006.

[10] R.V. Osorio, J.P. Lemos, T.W. Beech, G.G. Julian, and J.P. Chaumon. Scos-2000 release 4.0 : Multi-mission/multi-domain capabilities in esa scos-2000 mcs kernel. *Aerospace Conference, 2006 IEEE*, pages 1–17, 2006. doi: 10.1109/AERO.2006.1656141.

[11] Pascal Felber and Priya Narasimhan. Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. *Transactions of Computers*, 54(5):497–511, May 2004.

[12] Yansong (Jennifer) Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. Aqua: An adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–50, 2003. ISSN 0018-9340.

[13] Alexander Romanovsky. A looming fault tolerance software crisis? *SIGSOFT Softw. Eng. Notes*, 32(2):1–4, 2007. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1234741.1234767.

[14] Sumant Tambe, Jaiganesh Balasubramanian, Aniruddha Gokhale, and Thomas Damiano. MDDPro: Model-Driven Dependability Provisioning in Enterprise Distributed Real-Time and Embedded Systems. In *Proceedings of the International Service Availability Symposium (ISAS)*, Durham, New Hampshire, USA, 2007.

[15] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 241–244, 2002. ISSN 1527-1366.