ulm university universität uulm

Flexible and Reconfigurable Support for Fault-Tolerant Object Replication

Dissertation

zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik
der Universität Ulm

vorgelegt von

Hans Peter Reiser
aus Passau

2006

Amtierender Dekan:	Prof. Dr. Helmuth Partsch
Gutachter:	Prof. Dr. Franz J. Hauck
Gutachter:	Prof. Dr. Peter Dadam
Tag der Promotion:	15. Dezember 2006

Abstract

Using object-based middleware infrastructures is popular for the development of services in distributed systems. Active object replication is a suitable strategy to increase the availability of such services. Existing replication architectures, however, show some deficiencies in their flexibility and reconfigurability.

This thesis presents the FT*flex* architecture, which extends the CORBA-based Aspectix middleware with flexible and reconfigurable replication mechanisms. The contributions of this thesis support the development of replicated objects, enable a deterministic execution of object methods with multiple threads, and provide a flexible and dynamically reconfigurable group communication system.

First of all, this thesis designs an efficient replication architecture on the basis of fragmented objects. The presented solution offers transparency for clients and allows the relocation of parts of the service functionality to the client side. The developer of a replicated object can use semantic annotations to optimise the replication strategies. A tool for automated code generation simplifies application development.

Second, this thesis addresses the concurrent execution of object methods. The FT*flex* architecture provides new strategies that ensure determinism even if multiple threads want to access the shared object state concurrently. FT*flex* uses source code analysis and transformation as a novel way to influence the thread coordination in the replica code.

The third contribution of this thesis is a reconfigurable, consensus-based group communication system, which is used as a basis for active replication. This system supports variable failure models, ranging from crash-stop to Byzantine. A significant advantage of the system is its transparent support for consistent reconfigurations at runtime. In addition, the system offers internal instrumentation as a basis for autonomous adaptation and self-optimisation.

Zusammenfassung

Bei der Entwicklung von Diensten in verteilten Systemen erfreuen sich objektbasierte Middleware-Infrastrukturen einer großen Beliebtheit. Aktive Objektreplikation ist eine geeignete Strategie, um die Verfügbarkeit solcher Dienste zu erhöhen. Bei existierenden Replikationsarchitekturen lassen sich allerdings Defizite bei der Flexibilität und der Rekonfigurierbarkeit identifizieren.

Die in dieser Arbeit vorgestellte FT*flex*-Architektur ergänzt die CORBA-basierte Aspectix-Middleware um flexible und rekonfigurierbare Replikationsmechanismen. Die Beiträge dieser Arbeit unterstützen die Entwicklung von replizierten Objekten, ermöglichen eine deterministische Ausführung von Objektmethoden mit mehreren Threads und stellen ein flexibles und dynamisch rekonfigurierbares Gruppenkommunikationssystem bereit.

Zunächst entwirft diese Arbeit eine effiziente Replikationsarchitektur auf der Basis von fragmentierten Objekten. Die vorgestellte Lösung bietet Transparenz für Dienstnutzer und ermöglicht es, Teile der Dienstfunktionalität auf Nutzerseite zu verlagern. Der Entwickler eines replizierten Objekts kann semantische Annotationen verwenden, um die Replikationsstrategien zu optimieren. Ein Werkzeug zur automatischen Codeerzeugung vereinfacht die Anwendungsentwicklung.

Als zweites befasst sich diese Arbeit mit der nebenläufigen Ausführung von Objektmethoden. Die FT*flex*-Architektur stellt neue Strategien bereit, mit deren Hilfe Determinismus beim konkurrierenden Zugriff mehrerer Threads auf den Objektzustand erreicht wird. Dabei verwendet FT*flex* Quellcodeanalyse und -transformation als einen neuen Weg, um die Threadkoordinierung im Programmcode zu beeinflussen.

Als drittes stellt diese Arbeit ein rekonfigurierbares, einigungsbasiertes Gruppenkommunikationssystem bereit, welches als Grundlage zur aktiven Replikation verwendet wird. Dieses System unterstützt verschiedene Fehlermodelle, von "crash-stop" bis hin zu Byzantinischen Fehlern. Ein wesentlicher Vorteil des Systems ist die transparente Unterstützung von konsistenten Rekonfigurationen zur Laufzeit. Durch interne Instrumentierung bietet das System zudem eine Grundlage für autonome Anpassung und Selbstoptimierung.

Acknowledgements

I wish to express my deepest gratitude to my advisor, Prof. Franz J. Hauck, for his invaluable guidance, his encouragement, and his friendship. I am also very grateful to my second advisor, Prof. Peter Dadam, for his helpful suggestions for improving the structure and the clarity of my dissertation. Furthermore, the additional members of the dissertation committee, Prof. Peter Schulthess and Prof. Michael Weber, deserve recognition for their comments.

I am grateful to all of my colleagues in the Aspectix research group, who have provided me with inspiration and encouragement. I especially thank Rüdiger Kapitza for his collaboration on several papers in which parts of the results of this thesis have been published. I owe my sincere thanks to all who have read and helped to improve the final version of this dissertation, in particular Jörg Domaschka and Holger Schmidt. Andreas I. Schmied deserves special acknowledgement for his technical assistance.

I would like to thank the people at the Distributed Systems Lab at Ulm University for providing a friendly working environment for the past two years. Furthermore, I want to thank the staff at the Department of Distributed Systems and Operating Systems at University of Erlangen-Nürnberg, where I have done a large part of my doctoral research in a pleasant atmosphere. I am particularly grateful to Prof. em. Fridolin Hofmann, who sparked my interest in distributed systems, and to Prof. Wolfgang Schröder-Preikschat for his stimulating criticism.

This dissertation would not exist without the support of my family and of many friends. I want to thank my wife Fiona and my children for their love, support, and patience. My brother-in-law Lars Vinx deserves recognition for all his comments that helped to improve my English grammar.

Contents

List of Figures

Chapter 1

Introduction

Context: Distributed Middleware Platforms

The development of information technology in the past has lead to wide-spread use of distributed computing systems. Software developers are faced with new challenges due to the complexity of such systems. The use of *distributed middleware platforms* is an established approach to reduce the complexity of the development of distributed applications. The term *middleware* denotes an infrastructure situated above the operating system that provides services to the distributed application. Its functionality includes abstractions and mechanisms for remote communication (e.g., remote method invocation), services for referencing and finding remote applications, and mechanisms to convert data between heterogeneous system components. Today, middleware functionality is provided not only by dedicated platforms such as CORBA. Frequently, it is an integral part of programming languages and execution environments, such as Java RMI and .NET Remoting.

Fault-Tolerance Middleware Support

One important challenge for distributed applications is the ability to tolerate the failure of parts of the system. The inherent difficulty in developing fault-tolerant distributed applications demands support by the middleware infrastructure. Fault-tolerant CORBA (FT-CORBA), which in 2001 has become an integral part of the OMG's core specification of CORBA [OMG01], is an important example for the integration of fault-tolerance support into existing middleware. Other frameworks provide fault-tolerance support independently from the core middleware infrastructure. For example, JBoss Cache [BWSS05] provides replication for the JBoss Java application server [FR03].

Autonomous Computing

Another important development of the past few years is towards autonomic computing systems [KC03, LNP$^+$03]. Such systems are characterised by a set of self-x properties, such as self-configuring, self-optimising, self-repairing, self-protecting, and self-documenting. Autonomous systems raise new challenges, as they are much more dynamic than traditional systems. Applications and middleware platforms need to support such dynamic behaviour.

1.1 Problem Statement

Missing in Existing Systems: Flexibility, Multithreading, and Run-Time Reconfiguration

Currently, many fault-tolerant middleware systems use proprietary infrastructures. Object implementations not designed for fault-tolerant replication need to be significantly modified to be used in such infrastructures. Portability between various fault-tolerance platforms is, in general, not supported at all. Furthermore, existing middleware systems offer little flexibility. Replicated objects are usually treated as remote "black boxes", which impedes efficiency optimisation such as using semantic knowledge to optimise access strategies and moving some object functionality to the client side. In addition, method invocations at replicated objects are, in most cases, executed with a single thread only. The few exceptions to this rule are restricted to a simple, lock-based coordination model. In addition, current fault-tolerant middleware typically targets on static systems, with only little provision for flexible configuration, runtime reconfiguration, and autonomous adaptation. It is thus questionable whether existing fault-tolerant middleware systems can meet the new challenges of autonomic computing systems.

1.2 Main Contributions

Summary

This thesis presents the FT*flex* architecture for the replication support in the distributed object-oriented Aspectix middleware. The architecture uses fragmented objects as a novel way for integrating fault tolerance into an existing middleware system. It offers flexibility in terms of supported fault models and mechanisms for achieving replica consistency. FT*flex* provides innovative approaches to efficient multithreaded execution of object methods. Furthermore, it offers novel consistency mechanisms that allow dynamic reconfiguration and autonomous adaptation at runtime.

Fragmented Objects

Fault-tolerance support is realised using the powerful fragmented-object model of the Aspectix middleware. Full replication transparency is achieved at the client side. No difference exists between accessing a replicated and a non-replicated object, and remote references to replicated objects can be passed transparently as reference parameters. Given a fragmented-object middleware, no proprietary modifications to the middleware itself are required to support fault-tolerant replication. In addition, the presented approach avoids unnecessary indirection steps that other fault-tolerance infrastructures require for each access to a replicated object. No other existing replication middleware provides all these advantages simultaneously. FT*flex* not only describes a new approach to fault-tolerance support in object-based middleware systems, but also validates the advantages of the fragmented-object model of Aspectix.

Code Generation and Semantic Annotations

FT*flex* provides a code generation tool that simplifies the development of replicated objects. This tool enables the developer to specify semantic annotations that influence code generation. This way, replication strategies can be better tailored to object-specific properties than it is possible in other replication systems.

Multithreading

The problem of multithreading in replicated objects is another topic of this thesis. Replica consistency usually requires deterministic behaviour, and multithreading is a source of nondeterminism that is difficult to handle. Multithreading, however, is essential to avoid potential deadlocks that can occur in a single-threaded execution model, and it allows better utilisation of computational resources. This thesis presents a detailed taxonomy for the problems and benefits of single-threaded and multi-threaded execution models.

Interception of Synchronisation Operations

FT*flex* supports multithreaded method execution in replicated objects. To avoid nondeterminism, the infrastructure must be able to control the scheduling of threads and thus must intercept all scheduling-related operations. Existing multithreading-enabled systems use interception at the level of operating system or virtual machine. This thesis proposes automated source-code analysis and transformation to intercept synchronisation operations at a level that is independent of low-level operating system or virtual machine.

Deterministic Scheduling

This thesis proposes novel approaches for handling the intercepted scheduling operations. It specifies four variants of a scheduling module that is responsible for obtaining deterministic behaviour. Unlike previous work in this area, the presented scheduler modules fully support the native Java synchronisation model, including reentrant locks, condition variables, and time-bounds on wait operations.

Flexible Failure Models

At the level of replica consistency management, the FT*flex* infrastructure supports a wide range of failure models. As a result, it not only covers the domain of *classic replication systems*, which usually can tolerate only benign crash failures, but also applies to the domain of *intrusion-tolerant systems.* In all scenarios, the infrastructure currently uses *active replication* with strict replica consistency. Other variants, such as passive replication and replication with weaker consistency guarantees, are not considered in this thesis, but could easily be added to the FT*flex* architecture.

Aspectix Group Communication System

The *Aspectix group communication system* (AGC) is an internal part of FT*flex*, which is used for managing replica consistency. The AGC uses an instance of a distributed consensus algorithm that can be selected for various failure modules, ranging from benign crash-stop failures to malicious Byzantine failures. The configurable design allows the FT*flex* infrastructure to execute applications with various degrees of fault-tolerance, without having to modify the application.

Configurable Consistency Strategies

For each failure model, the AGC provides algorithmic variants, allowing the system to be tailored to developer requirements, network properties, failure frequencies, and client interaction patterns. Due to the encapsulation of consistency strategies within the group communication system, replicated objects can be used in many environments without requiring changes to the replicated object or the remaining middleware. A *classification* of variant properties allows the selection of the best variant for given developer-defined parameters and observed system conditions.

Runtime Reconfiguration and Autonomous Adaptation

Another important contribution is the provision of mechanisms that allow dynamic reconfigurations. Large-scale reconfigurations, such as the replacement of the internal consensus algorithm, are supported in a way such that (a) strict replica consistency is guaranteed across the reconfiguration, (b) the reconfiguration is fault-tolerant (i.e., can tolerate concurrent failures), and (c) has only minimal impact on the running application (i.e., is significantly more efficient than simply shutting down the system and restarting it in a new configuration). In addition, autonomous self-optimisation is partially supported. For example, algorithms can be adjusted automatically to provide optimal performance given the current network properties, the availability of nodes, and client interaction patterns. Reconfigurability and adaptability increase the utility of the FT*flex* architecture in the context of dynamic systems.

1.3 Structure of this Thesis

This thesis is structured as follows:

Chapter 2: Background

Chapter 2 defines the context of the thesis by presenting the relevant background. It discusses the basic functionality of distributed object middleware. Existing approaches to fault tolerance in distributed systems are summarised. Special attention is given to replication support in object-oriented middleware and to the architecture of Fault-Tolerant CORBA. Finally, the chapter provides exact definitions of important terms and models that are used in this thesis.

Chapter 3: Existing Object Replication Systems

Chapter 3 discusses problems in existing object replication systems. It evaluates multithreaded execution of object methods. Furthermore, the chapter discusses existing group communication technology and outlines deficiencies that relate to this thesis. Finally, it presents an overview of the research goals; these can be divided into three categories, which are the subjects of the subsequent three chapters.

Chapter 4: Middleware Integration and Development Support

Chapter 4 proposes the realisation of fault tolerance in distributed systems using fragmented objects. It presents the design of the flexible and reconfigurable FT*flex* architecture, which adds replication support to the Aspectix middleware. Special emphasis is put on consistency issues that are caused by interactions between independent replicated objects. Code generation and semantic annotations are presented as means to simplify the development of replicated objects, to handle nondeterministic replica behaviour, and to automate state transfer between replicas. Finally, the chapter discusses management and dynamic reconfiguration of replica groups from a high-level point of view.

Chapter 5: Deterministic Multithreaded Execution

Chapter 5 discusses deterministic multithreading in replicated objects and introduces the Aspectix DEterministic Thread Scheduler (ADETS). It proposes source-code transformation as a new approach to intercepting native Java synchronisation mechanisms. The chapter presents four algorithms that ensure a deterministic scheduling of threads in replicas. Furthermore, an experimental analysis compares and evaluates all implemented scheduling strategies.

Chapter 6: Group Communication

Chapter 6 takes a closer look at the low-level layer of the FT*flex* architecture. It presents the Aspectix group communication system (AGC), which offers totally ordered multicast. The AGC is used for consistent active object replication. It provides algorithmic variants, which offer variability in terms of failure models and communication mechanisms. In addition, reconfiguration support and autonomous adaptation are implemented at this low-level layer.

Chapter 7: Conclusions

Finally, Chapter 7 presents concluding remarks. It summarises the contributions and the limitations of this thesis and discusses open problems and challenges for future work.

Chapter 2

Background

This chapter describes the wider scope of this thesis. It discusses the basics of distributed object-orient programming and gives a review of strategies to provide fault tolerance in distributed systems. Furthermore, it defines the system model that is used in this thesis.

2.1 Distributed Object Middleware

Object-Oriented Programming

Object-oriented programming has a long tradition in computer science, starting in the 1960s with the programming languages Simula [DN66] and Smalltalk [Kay93]. Today, it is one of the most popular programming paradigms, used in languages such as C++ and Java. State (data) and behaviour (functionality) are encapsulated in *objects*, which are accessed via *methods* of an *interface*. The programming model provides abstraction, encapsulation, inheritance, and polymorphism [Weg90]. The strength of object orientation is based on its ability to separate interfaces from implementations, to substitute various object implementations that have the same interface, to promote reusability and modularisation, and to support hierarchical decomposition.

Distributed Computing

Distributed computing is a useful model for many applications, and, along with the development of networked systems, is steadily rising in popularity. Distributed applications are frequently structured on the basis of a client-server model. In this model, a client requests a service operation from a remote server and receives the operation result. Distributed object-oriented systems apply the principle of object-oriented programming to distributed applications.

2.1.1 Distributed Object-Oriented Programming

Distributed Object-Oriented Programming

In distributed object-oriented programming, a service is modelled as an object with a public interface, and clients interact with the server using remote method invocations. Ideally, such a remote invocation is fully transparent to caller and callee; that is, no difference exists between a local and a remote method invocation. Transparency is only partially achieved in practice. For example, caller and callee may fail independently in a distributed system, which can lead to situations that cannot occur locally. Furthermore, the data types that can be passed as invocation parameters face limitations in the distributed case, as local file handles, local memory references, etc., cannot easily be transferred to a remote host.

Interface Definition

The interface of a remote object is defined either using programming-language constructs or using another special interface definition language. Usually, the client uses synchronous remote invocations. This behaviour is identical to that of local invocations: the client issues a remote invocation and, after that, waits until it receives a response from the server that hosts the remote object.

Invocation Semantics

The semantics of remote method invocations can be characterised by the behaviour in failure situations. Spector [Spe82] provides the following classification for invocation semantics[1] .

- *maybe (best effort)*: In optimal executions, the remote method is executed once, and the client receives a reply. If any failure occurs, including lost packets at the network level, the remote method may or may not be executed, and the client may or may not receive a reply.
- *at-least-once*: Invocations are re-tried for as many times as it is necessary to get a reply. This semantics is able to mask communication failures. The call, however, may be executed multiple times at the servant. In case of a client failure, the call may or may not be executed.
- *last-of-many*: This is an extension of *at-least-once*, which guarantees that the reply that the client receives belongs to the last invocation.
- *at-most-once*: Invocations are executed at most once. In this semantics, calls may be re-tried after communication failures. The servant guarantees that it does not execute the invocation a second time, but instead retransmits the previously obtained result. If the invocation is not successful due to a failure, the servant method may have been executed zero or one times.
- *exactly-once*: This semantics guarantees that the remote method is executed exactly once, in spite of all failures. It requires that the servant be able to recover after failures. Exactly one execution in case of client failure is guaranteed only if the client is able to recover as well.

In a distributed system, *exactly-once* semantics are hard to achieve. It requires transactional behaviour of the servant. As the implementation of *exactly-once* semantics causes high runtime costs, weaker variants are used in practice. CORBA and Java RMI, for example, provide an *at-most-once* semantics.

Stubs, Skeletons, and Servants

Remote method invocations are usually implemented on the basis of middleware infrastructure. At the client side, the caller uses a *stub* to access the remote service; this stub has the same interface as the remote service and, in general, is automatically created on the basis of the interface definition. The stub maps remote method invocations to messages that are sent to the remote host, and maps incoming result messages to a value that is returned to the caller. Method parameters are transformed into a transferable network format and vice versa, a process called *marshalling* and *unmarshalling*. For this purpose, the stub uses functionality that is directly provided by the middleware system. The object that implements the remote service at the server is called the *servant*. A

[1]Instead of Spector's terms "only-once-type-1" and "only-once-type-2", the more common terms "at-most-once" and "exactly-once" are used respectively.

skeleton or *server-side stub* maps incoming messages to method invocations at the servant and passes the invocation result back to the client.

Nested Invocations

Clients and servers are not necessarily completely disjunctive entities. Rather, it is possible that the server issues further remote invocations during the execution of a remote method. In this case, the server acts as a client for these invocations. Such invocations are called *nested invocations*.

References and Binding

Distributed object-oriented systems need unique references to distributed objects. These addresses are globally valid. A client has to *bind* to the remote object using this address to get access to the object. The binding operation loads the corresponding stub code at the client and initialises the stub with location information obtained from the reference data. Binding operations can happen explicitly and implicitly. In an *explicit binding operation*, the client application itself asks the middleware system to convert an external representation of a remote reference into a local reference to a stub. In contrast, in an *implicit binding operation*, the stub is loaded automatically without a dedicated application request. Implicit binding usually happens if the application receives the remote reference as a marshalled parameter or return value of a remote invocation. Only such an implicit binding allows transparent reference passing across host boundaries, identical to what is locally possible in object-oriented programs.

2.1.2 Popular Middleware Systems

CORBA

CORBA (Common Object Request Broker Architecture) is an open and vendor-independent middleware architecture for distributed object-based applications, defined by the OMG (Object Management Group) [OMG04]. It provides basic mechanisms for remote method invocation through an ORB (Object Request Broker). Interfaces are defined in CORBA IDL (interface definition language). CORBA allows for interoperability between CORBA-based systems implemented in various programming languages, or running on ORBs of different vendors.

.NET Remoting, Java RMI, Web Services

Other middleware systems work in a similar way. Java RMI supports remote method invocations in the Java programming language [Sun04]. The main difference to CORBA is the tighter language integration, which lacks support for cross-language interactions, but improves transparency and simplifies the system. Microsoft's .NET Remoting is a similar approach, which is simpler than CORBA but still allows cross-platform interoperability by defining a binary data exchange format [MWN02]. Web services are another approach to interoperable network-based applications [ACKM03]. They define interfaces in a public format such as the *web service definition language* (WSDL), and use a client-server structure for interaction on the basis of a remote invocation protocol such as SOAP [CDK$^+$02]. All such middleware systems use an RPC-based interaction for invoking object methods at a remote location.

2.1.3 Aspectix

Overview of Aspectix

Aspectix is a middleware based on the fragmented-object model, which aims at providing superior flexibility and adaptivity compared to other existing middleware systems [HBG$^+$01, RHKS03]. It supports heterogeneous systems, provides

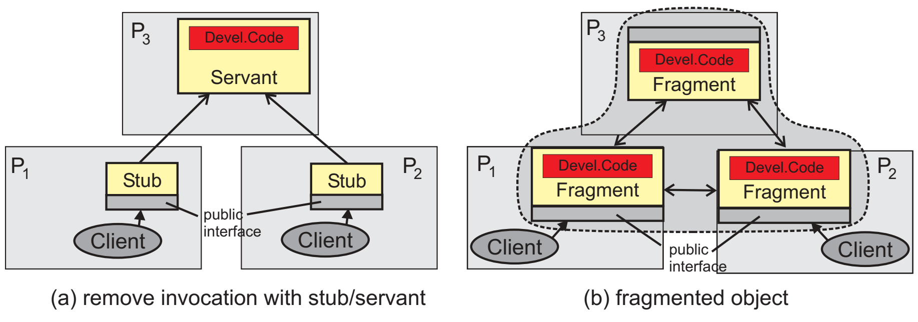


Figure 2.1: Comparison of client-server model and fragmented-object model

mechanisms to control the quality-of-service of applications, and facilitates dynamic reconfigurations at runtime.

Fragmented-Object Model

The concept of fragmented objects has first been used by FOG [MGNS94] and then by Globe [HvDvS$^+$95]. It offers a distribution model that is more flexible than the traditional RPC-based client-server model. An object is no longer associated with a single location; the interaction from the client side is no longer restricted to a remote invocation via a client-side stub. Instead, the object itself is an entity that spans multiple nodes. Each client that interacts with the object requires a local *fragment*; all fragments are conceptionally part of the object (see Figure 2.1). A fragmented-object middleware infrastructure allows arbitrary distribution of object functionality and state on the fragments as well as arbitrary inter-fragment communication.

IOR References to Fragmented Objects

Aspectix supports this object model within a CORBA-compliant middleware infrastructure. It uses CORBA *interoperable object references* (IORs) to reference fragmented objects. It provides client-side transparency by automatically loading an object-specific local fragment instead of a standard stub when the client binds to such an IOR.

Quality of Service

A special focus of Aspectix is on the support for quality-of-service awareness. The code that is loaded for a fragment can be selected individually from a set of available implementations. This selection can also be changed at runtime by transparently replacing the fragment implementation; as a result, systems can adapt themselves to provide the desired quality-of-service. Thus, Aspectix offers a flexible platform for dynamic reconfigurations.

Automated Code Generation

If the object developer had to implement all fragment implementations manually, the flexibility of the fragmented-object model would be outweighed by a development complexity much higher than in the simple client-server model. In practice, a fully manual implementation of fragment code is not necessary. The Aspectix middleware provides a code generation tool that is able to automate the creation of most fragment code for standard tasks. Only the specific object functionality needs to be implemented by the developer.

Relevance of Aspectix Features to Object Replication

In the context of this thesis, the Aspectix infrastructure provides important mechanisms to simplify the support for flexible and reconfigurable object replication. The fragmented-object model of Aspectix transparently enables the loading of object-specific fragment code at the client side. This advantage is useful for loading tailored functionality for accessing a replica group. The code generation tools of Aspectix can support the flexible provision of object-specific replication mechanisms. Furthermore, adaptivity and reconfigurability is also inherently supported by Aspectix middleware mechanisms.

Outside CORBA

The Aspectix approach for supporting fragmented objects is not limited to CORBA. Our FORMI architecture [KKSH05, KDH$^+$06] demonstrates that the same concepts can be used for Java RMI. Our replication prototype for .NET Remoting [RDH05] shows that this environment also offers a similar extensibility. This thesis concentrates on the support for object replication in the context of CORBA-based distributed applications. The referenced prototypes show that the FT*flex* architecture is also applicable to non-CORBA systems.

2.2 Fault Tolerance in Distributed Systems

Failures in Distributed Systems

Distributed systems are permanently faced with the problem of node failures. A famous quote by Lamport puts it this way: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable” [Lam87]. This problem can be addressed by hardware-based mechanisms that reduce the probability of failures, by mechanisms that support system recovery after failures, and by transparently masking the effect of failures.

Reliable Hardware

Dependability can be increased using special highly-reliable hardware (e.g., using error-correcting memory, redundant storage, or redundant power supplies). This approach is typically used for applications with demanding availability requirements in mission-critical systems. An increase in reliability comes along with significantly increased hardware costs. Software-based mechanisms for fault tolerance are sometimes seen as a way to reduce these costs. It is, however, more appropriate to consider hardware-based approaches as fully orthogonal to software-based approaches. Both concepts can independently be used at the same time to increase the dependability of a system. Hardware-based approaches to fault tolerance are not discussed in this thesis.

Checkpointing and Recovery

Recovery is usually supported by the periodic creation of consistent checkpoints, which can be used to reset the system to this state after failures. Although this approach has the advantage of minimal overhead in failure-free operation, in case of failures it causes a loss of all state modifications since the last checkpoint. In addition, it is faced with a period of unavailability while the system recovers. It requires accurate and timely failure detection. Falsely suspecting a failure causes unnecessarily high recovery costs and state loss; using long timeouts for failure detection decreases the risk of false suspicions, but increases the time until the system recovers and resumes operation. Given an efficient checkpointing implementation, recovery-based fault tolerance is feasible in practical systems. For example, Plurix [SFGS04] provides an execution model on the basis of restartable transaction with an optimistic synchronisation scheme. The system handles node failures with a simple and fast reset of the cluster, after which the execution of transactions may continue from a consistent distributed state.

Replication

Replication is another approach to fault tolerance in distributed system. The functionality of an entity is redundantly made available on multiple nodes. After the failure of a node, the remaining replicas can still provide the full functionality and can thus transparently mask the effect of the failure. Usually, the replicated entity has some state that has to be kept consistent. Replication can be used at several levels of the distributed system. This section discusses the replication of file systems, databases, and objects.

2.2.1 File System Replication

File System Replication

From a client point of view, a replicated file system appears as a single logical remote file system. Internally, the replicated file system stores files redundantly on the local file systems of multiple nodes.

Read-Only Replication and Weak Consistency

Read-only replication and weak consistency strategies are popular in the domain of file systems. In read-only replication, copies from a primary source are made on secondary nodes to support load-balancing; data modifications are only possible on the primary data source and lack fault tolerance. Solutions in this category are, e.g., the replication support in the Andrew File System [Kaz88] and in SharePlex [Uy02]. In optimistic replication strategies, modifications are made without synchronisation, and updates are later propagated to all replicas. If inconsistencies are detected, manual conflict resolution by the user becomes necessary. Prominent examples for optimistic replication are Locus [WPE$^+$83], Coda [SKK$^+$90], and Ficus [PGJH90].

Strong Consistency: Update-All, Primary-Copy, and Consensus Approach

Only file systems with strong replica consistency are comparable to the replication mechanisms this thesis focuses on. An *update-all strategy* ensures that all modifications are made in consistent order on all replicas. For example, Deceit [MS88] uses this approach based on the totally ordered group communication provided by ISIS [BJ87]. A *primary-copy approach* is, for example, used by the Harp File System [LGG$^+$91]. In this system, all backup replicas have to acknowledge all modifications. A majority-based view-change algorithm excludes crashed backups and, after a primary crash, transparently selects a new primary. *Distributed consensus algorithms* can also be used to replicate parts of a distributed file server. For example, Frangipani [TML97] uses a lock server that provides multiple-reader-single-writer locks to coordinate concurrent access to replicated disk data. The lock server is fully decentralised to obtain scalability and fault tolerance; it uses Lamport's Paxos algorithm [Lam89] for consistent replication of the global lock server state.

Peer-to-Peer-based Replicated File Systems

Recently, decentralised distributed file systems on the basis of structured peer-to-peer infrastructures have become popular, such as OceanStore [KBC$^+$00] (based on Tapestry [ZHS$^+$04]), CFS [DKK$^+$01] (based on Chord [SMK$^+$01]), and PAST [RD01a] (based on Pastry [RD01b]). File systems based on peer-to-peer systems store files as sets of (key, data) blocks in the base systems; these data blocks are read-only and may arbitrarily be cached and replicated without provisions for consistency. The only exception to this rule is a *root* object that identifies the set of blocks belonging to the most recent version of a file. The root information needs to be replicated for fault-tolerance reasons; usually, majority voting techniques are used make consistent updates on a group of nodes. OceanStore, for example, uses Castro's Byzantine variant of the Paxos algorithm [CL99] for consistency of the root information.

2.2.2 Database Replication

Database Replication

Distributed database systems frequently use replication techniques. Often, these techniques are justified primarily by efficiency reasons. Replication increases scalability and improves response time, as more replicas balance the load and handle more read requests in the same amount of time. The second purpose of replication in database systems is to increase availability. The following literature review concentrates on the fault-tolerance aspect of database replication, as this aspect corresponds to the focus of this thesis.

Replica Control and Consistency Models

Similar to all other replication systems, the challenge for replicated databases is to control the replicas such that they maintain a consistent shared state. Existing approaches can be classified into *primary-copy*, *read-one-write-all*, and *quorum-based consensus*. A more detailed discussion of approaches can be found in Bernstein et al. [BHG87].

Primary-Copy Approach

In a *primary-copy* approach [AD76, Sto79], an update is first processed by a primary node. The primary then updates the state of the secondary replicas. After a primary failure, a secondary is selected to become the new primary. Efficiency reasons usually advocate an implementation in which the primary updates all other copies only asynchronously. The non-primary copies may lag behind with updates; they are unable to guarantee that read operations access the most up-to-date state. Using local read operations on secondary replicas is only possible with a weaker semantic model: instead of reading consistent up-to-date values, snapshot semantics is provided, which ensures consistency of the values read, but permits reading outdated values. Consistency may, for example, be defined by a versioning concept [MPL92].

Read-One-Write-All Approach

The *read-one-write-all* (ROWA) strategy achieves consistency of replicas by updating them all on each state modification. This way, all replicas always have an up-to-date state, and read operations can be done locally on any replica. For read-only requests, this replication strategy offers optimal performance. In contrast, the efficiency of write transactions is determined by the slowest node in the system. If one node becomes unavailable, no modifications can be made. This problem is reduced in the similar read-one-write-all-available (ROWAA) approach [BG84]. In ROWAA, a crashed replica is not updated; after the replica recovers, it needs to update its state before it may handle any requests. This strategy requires accurate failure detection.

Quorum Consensus Approach

In a *quorum consensus* strategy, all write transactions need to be executed at a write quorum W of nodes, and all read transactions have to be executed at a read quorum R of nodes. Any two write quorums W_1, W_2 and any pair of read quorum R and write quorum W need to intersect in at least one element. The intersection ensures that at least one node with the most recent state is part of the quorum. All transactions will use this up-to-date state, which can be identified by state revision numbers that are increased on each update or by logical time stamps of the last modifications. An early example of quorum consensus was the majority voting mechanism introduced by Thomas [Tho79], in which all nodes are equal and numerical majority is used to define both read and write quorums. More flexible approaches use weighted voting for replicated data [Gif79], or other generalisations such as crumbling wall quorums [PW95]).

Group Communication

The use of group communication technology for replicating databases is advocated by several researchers. Kemme et al. [KA98] argue that a tight integration of database replication strategies with low-level group communication protocols leads to systems that are more fault-tolerant than primary-copy (in which the primary is a single point of failure) and which are more efficient than traditional ROWA strategies, while maintaining the same transactional semantics found in centralised systems. Pedone et al. [PGS98] argue that the use of atomic broadcast primitives increases the efficiency of replicated databases with deferred replica updates. More recently, Wiesmann [Wie02] has provided a detailed discussion of the benefits from using group communication for database replication. This means that group communication technology can be used successfully for the consistent replication of distributed databases.

2.2.3 Object Replication

Object Replication

Object-based middleware uses *objects* as units of distribution (see Section 2.1.1). An object at this level typically is a coarse-grained entity; the term *service* is synonymously used for it. The (logical) distributed object can be composed of multiple local objects at the programming-language level. The distributed object has an explicitly defined interface, which is used as the basis for remote invocations. An object replication system provides redundant instances of an object at multiple nodes; both state (data) and functionality (code) of the object are replicated. *Passive replication* and *active replication* are two fundamental approaches to consistency management of replicated objects [CDK94].

Passive Replication

In *passive replication* (also called *primary-copy replication*), requests are processed by a single primary replica. The primary is responsible for updating the state of the secondary replicas. Usually, the state transfer is not triggered immediately after each state modification, as doing so would be too expensive. Instead, updates are sent asynchronously in larger time intervals. If the primary fails, one secondary replica needs to be selected as new primary. The secondary, however, may lag behind with state updates compared to the failed primary. In this case, the new primary has to re-execute all client operations that happened after the last state transfer. For this purpose, a request log stores all client requests that the primary has processed after the last successful state update of all secondary nodes.

It is common to use secondary replicas only as backup nodes. Using them for read-only operations is rarely done in practice as (a) the middleware often does not know whether operations are read-only, and (b) secondary replicas can lag behind with state updates, which results in a weaker semantics, and it is unknown whether such a semantics is acceptable for clients.

Active Replication

In *active replication*, all replicas independently process the same set of client requests. The term *state-machine replication* is also used for this approach [Lam78]. The effect of executing a set of methods is assumed to be deterministic. As a result, the state of the replicas remains consistent. The simplest strategy is to assume a deterministic behaviour of each single method and to execute all methods sequentially in total order. As all replicas keep an up-to-date object state, masking the failure of a replica remains fully transparent to clients. All replicas execute the method requested by the client; the client may receive a reply from one available replica without any delay, even if other replicas fail.

Semi-Active Replication

Hybrid approaches that combine features of both replication styles are advocated by some authors. In the Delta-4 project, Powell et al. [PCD90] introduced the notion of *semi-active replication*. This replication style aims at combining the fast recovery of active replication with relaxed determinism requirements. All replicas autonomously execute all deterministic computations. One replica is selected as leader and becomes responsible for all nondeterministic decisions. The leader's decisions are transferred to all other replicas as a kind of "mini-checkpoint".

Semi-Passive Replication

Semi-passive replication was defined by Défago et al. [DSS98] as a variation of passive replication. A rotating coordinator paradigm is used to select one node that processes a request. In ordinary passive replication, a new primary is selected only after the previous primary is suspected to have crashed. The suspected primary is excluded from the replica group; in case of a false suspicion, it has to re-join the replica group in a costly operation. In semi-passive replication, the primary role may quickly be moved from one node to another based on aggressive timeouts, while a second, conservative timeout is used for replica group membership. Furthermore, clients send their requests to all replicas, instead of sending them only to the primary as in passive replication. This strategy eliminates the need to re-send the request after a primary failure.

Nested Invocations

Similar to non-replicated distributed objects (see Section 2.1.1), a replicated object A can invoke a remote method of a replicated object B. Supporting such invocations adds complexity to the replication infrastructure; this is why only some existing systems provide the necessary support. For nested invocations, the infrastructure has to make sure that the remote method at object B is invoked only once, even if multiple replicas issue this invocation (due to active replication or failover to a secondary in passive replication). In the other direction, the result of the invocation at B has to be delivered to multiple replicas of object A.

Group Communication

Group communication is frequently used for replication strategies. Most important of all, the usual implementation of active replication uses totally ordered multicast for delivering requests to all replicas in total order. Semi-active and semi-passive replication also require a multicast mechanism to send requests to all replicas. Group communication protocols provide such multicast facility with the desired semantics.

2.2.4 Summary of Replication Technologies

Manual Conflict Resolution

The possible approaches to replication are similar for all discussed replication domains. Approaches with uncoordinated updates, which potentially require manual intervention in case of conflicts, are most popular for file systems. In a few systems, this method is also used for database replication (e.g., Garcia-Molina et al. [GMAB$^+$83]). Manual conflict resolution is only feasible if conflicts happen only rarely in practice and if users have sufficient knowledge to manually resolve conflicts. With the rising complexity of distributed systems, such an approach will become less and less feasible in most scenarios.

Read-only and Backup Replicas

Primary-backup systems exist in several variants. In the simplest form (e.g., passive replication of objects), the replicas serve only a backup role, taking over the role of the primary after primary failure. Often, backup replicas are used to handle read-only operations, which allows load-balancing. In this case, the update strategy influences the read semantics. Primary-backup systems either use a synchronous update of all replicas on each state modification, or offer a weaker semantics that permits read operations on replicas that do not have the most recent state. Modifications always have to access the primary. If the primary fails, a failover to a backup replica is necessary to restore the ability to modify the object state.

Consensus-based Strategies

Consensus-based strategies eliminate the need for a coordinating primary. The lack of dependency on a single node results in a faster reaction to failures; ideally, node failures are fully hidden from users. In contrast, in a primary-backup scheme the transition to a new primary after a primary failure results in

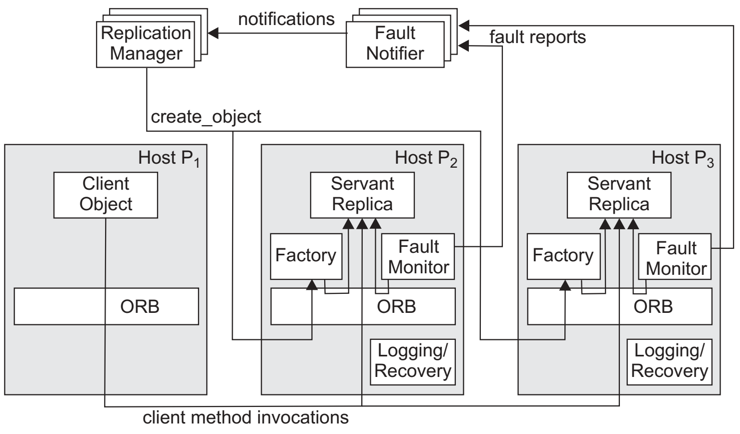


Figure 2.2: Architecture of Fault-Tolerant CORBA

a period of system unavailability. Consensus-based strategies also avoid the dependency on accurate failure detection. This separation is important for asynchronous systems that do not permit accurate failure detection. A drawback of consensus-based strategies is that they typically have higher overhead than primary-copy approaches.

Focus of this Thesis

The focus of this thesis is on consensus-based strategies for active replication of distributed objects. The FT*flex* architecture realises this replication style on the basis of a totally ordered group communication system. The group communication system integrates consensus-based strategies for consistency management. The FT*flex* architecture provides strict replica consistency and does not consider weaker consistency models. The infrastructure offers mechanisms to select and configure the consensus protocol for optimal efficiency for a given execution environment and application.

2.3 Fault-Tolerant CORBA

Fault Tolerance in CORBA

Initially, the CORBA platform had no support for fault tolerance. Electra [Maf95], Eternal [MMSN98], and Maestro [VB98] are early examples of systems that add fault tolerance to CORBA. In 2001, the OMG added the fault-tolerant CORBA specification (FT-CORBA) to its core CORBA specifications [OMG01]. FT-CORBA defines a management architecture and basic interfaces, but leaves details like consistency protocols to the concrete middleware implementations.

2.3.1 Management Architecture

Overview of FT-CORBA

The FT-CORBA standard describes a system architecture that is composed of the *replicas*, *factories* that can create replicas, the *replication manager*, and *clients*. In addition, services for *request logging*, *checkpointing/recovery*, and *failure detection* can be deployed. The overall architecture is shown in Figure 2.2. All components are themselves implemented as CORBA objects.

Replication Manager

The *replication manager* is responsible for managing multiple replicated objects within a replication domain. Typically, the replication manager is replicated to avoid a single point of failure. The replication manager is responsible for managing properties (such as replication style and the required number of replicas), for the creation of replica groups, and the management of replicas (e.g., the addition of new replicas to a replica group).

Replica Factories

To create replicas, a *factory approach* is used. A factory needs to be provided on each node on which a replica shall be placed. The factory offers methods to remotely create objects. Factories are plain CORBA objects and can be used remotely by the group manager.

Replica Implementation

The replica implementation is basically the same as a non-replicated CORBA servant implementation. No difference exists for stateless services. For stateful services, the replicated object needs to implement an additional interface for state transfer.

2.3.2 References to Replication Groups

CORBA IOR

CORBA uses *Interoperable Object References* (IORs) to reference remote objects. Internally, the IOR is composed of a set of profiles. Each profile specifies a way to contact the object. In addition, *tagged components* can be stored in a *multiple component* profile to provide access-independent information, such as a unique object ID. Plain COBRA objects, which can be accessed via IIOP, use an IIOP profile in the IOR.

FT-CORBA IOGR

Replication requires that references point to a group of objects instead of pointing to a single object on one node. FT-CORBA defines *Interoperable Object Group References* (IOGRs) for this purpose. An IOGR is an IOR that contains a separate IIOP profile for each replica. In addition, the IOR contains a global object ID and a reference version number.

Limitations

Using a special form of the IOR as a group reference permits a limited degree of interoperability with non-fault-tolerant CORBA systems. Such a system will iterate over all IOR profiles to find a contact address that allows remote access. A non-fault-tolerant client will find an available contact replica even if some of the replicas have failed. Only partial fault tolerance is achieved by this approach. Failures that occur after initially finding a contact replica are not handled transparently. If the client runs on a fault-tolerant ORB, it will provide additional functionality, for example, to repeat the invocation after a failure.

Reference Updates

During the lifetime of a replicated object, new replicas can be added to the replica group or can substitute crashed ones. Such group changes invalidate the information in the IOGR. Each modification to the replication group membership creates a new version of the IOR. On each client interaction, the replicas can inform the client about new versions of the IOR, which enables the client to obtain the most recent contact information of the group.

2.4 System Model

Definition

This thesis uses Bal's definition of a distributed computing system.

Definition 2.1 (Distributed System) *"A computing distributed system is composed of several autonomous processors without shared main memory, but cooperating by message passing over a communication network" [Bal90].*

Notation

Processors are denoted by the symbols $P_1, P_2, \ldots$; the terms *node* and *process* are used as synonyms for processor. The connections of the communication network are called *channels*. Nodes exclusively interact by the exchange of messages; no other way for exchanging information exists. This definition not only excludes systems with shared main memory, but also shared disk storage that is accessible from multiple nodes.

System Models

There are various system models that restrict the behaviour of processes and communication channels. The *synchrony model* defines timing assumptions about the behaviour of nodes and channels. The *failure model* defines the behaviour of components that fail.

2.4.1 Synchrony Model

Basics of Synchrony Models

The synchrony model concerns two parameters. First, it defines bounds on the execution speed of processors (i.e., how long it takes at most for a node to execute a step). Second, it defines bounds on the message transmission delay (i.e., the amount of time that elapses between the emission and the reception of a message). The two extremes of the spectrum of synchrony models are given by the *synchronous* and *asynchronous* system model.

Synchronous System Model

Definition 2.2 (Synchronous System Model) *A distributed system is* ***synchronous*** *if known upper bounds exist for the execution time of each local computation and for the messages transmission delay.*

The strict timing assumption of the synchronous model provides a convenient programming model. For example, algorithms can be executed in synchronous rounds, and real-time clocks can be synchronised with a known upper bound on the synchronisation error. Often, the synchronous model permits simple and efficient algorithms that are not possible with other, less strict synchrony models.

Asynchronous System Model

Definition 2.3 (Asynchronous System Model) *A distributed system is* ***asynchronous*** *if no concept of physical time is used. The duration of local computation steps and message transmissions is unbounded.*

The asynchronous system model makes no timing assumptions at all. It is the most general synchrony model. Algorithms for the asynchronous system model work in any synchrony model.

Comparison

Although the synchronous model provides a convenient programming abstraction, it makes assumptions that are hard to maintain in a realistic environment. For example, an Internet-based distributed application has no strict guarantees on message delivery delay. On the other hand, the asynchronous model is convenient because of its generality. Several important problems, however, such as the distributed consensus problem [FLP85], cannot be solved in an asynchronous model. In other words, neither the synchronous nor the asynchronous model is adequate for most realistic distributed systems.

Intermediate Models

In practice, intermediate models with some partially synchronous assumptions have to be used. For example, processes might know an approximate time bound for local computations or for message transmission delays [Lyn96], or such bounds might exist but be unknown to all processes [DLS88]. A frequently used model is that the system is basically synchronous, but that the bounds of the synchrony model may be violated to some extent. If those violations were not restricted at all, this model would be equivalent to the asynchronous model. The *eventually synchronous model*[2] provides a practical restriction [DLS88].

Eventually Synchronous Model

Definition 2.4 (Eventually Synchronous Model) *A distributed system is* ***eventually synchronous*** *if there exists an unknown time T, after which the execution time of computations and the message transmission delay is bounded by a known upper limit.*

Algorithms for the eventually synchronous model are similar to those for a synchronous model, as they can use the timing properties that are guaranteed to be valid after time T. It is, however, necessary that such algorithms be able to tolerate arbitrary violations of the timing assumptions before T.

Oracle Approach

Other systems use an *oracle approach.* The system is assumed to be completely asynchronous, but an oracle provides some additional information. For example, Chandra and Toueg [CT96] propose failure detector oracles to solve the distributed consensus problem in an asynchronous system. Both reliable failure detection and distributed consensus are problems that are impossible to solve in a fully asynchronous system. With an oracle that gives information about crashed nodes (with only weak accuracy semantics), distributed consensus becomes solvable.

Oracle-based Algorithms in Multiple Environments

This approach enables the use of an algorithm in multiple environments. The correctness verification of the algorithm does not depend on the specific assumptions of a partially synchronous system model provided by the environment. Instead, the verification can be based on the abstract properties of the oracle. All that is necessary is to provide an implementation of the oracle for each environment in which the algorithm shall be used. The algorithm will work correctly in the environment if the abstract oracle properties can be verified given the specific properties of the real environment.

2.4.2 Failure Model

Overview

Components of a distributed system may fail in various ways. It is important to define which kind of failures are expected to occur in the distributed system; providing such a definition is the task of the *failure model.* If nodes fail in a way not covered by the failure model, a complete system failure can occur.

Node Failures

Hardware and Software Failures

Node failures can be attributed to hardware or software. Hardware may fail due to reasons such as power failures, wear out, or design failures. Software failures can happen in the operating system, the middleware, or the application itself. From the point of view of the distributed system, it is not necessary to distinguish between failures caused by software or hardware; all that is relevant is that a node is no longer functioning correctly. The purpose of fault tolerance is to maintain the functionality of the system even if some nodes are faulty.

[2]Throughout this thesis, the term *eventually* is used in its strict mathematical sense. An event happens *eventually* if it definitively happens within an unknown but finite amount of time.

Number of Tolerable Failures

Functionality of a distributed system can be maintained only if sufficiently many nodes are functioning correctly. In the extreme, the distributed system will certainly fail if all nodes are faulty. It depends on the protocols and the system model how many correct nodes need to be available to ensure proper system operation.

Independent Failures

In the design of a fault-tolerant application it is important that reasonable measures are taken to make the failure of nodes independent. It would be fatal if a single fault caused the failure of multiple or all replicas of a service. This is more likely to be a problem with software failures (e.g., a bug in a service implementation that manifests itself on all nodes) than with hardware failures. Frequently, the same operating system, middleware system, and application will be used on all nodes. One approach to tackle this problem is to use *multi-version programming*, where independent implementations of these entities are used in different nodes. Multi-version programming is not a main subject of this thesis, but it can easily be combined with the presented replication architecture.

Failure Model

A further distinction can be made on the basis of the behaviour of faulty processes. In a real system, arbitrary behaviour of faulty nodes may be expected. If such arbitrary behaviour is considered to be sufficiently unlikely, different failure models with restrictions on the behaviour of faulty nodes can be used; such restrictions usually result in simpler and more efficient protocols. The two extremes of failure models are *crash-stop* and *Byzantine*. Several intermediate models may be used; important representatives defined in this section are the models *crash-recovery* and *restricted Byzantine*.

Crash-Stop Failure

Definition 2.5 (Crash-Stop Failure Model) *In a crash-stop failure model, a* ***correct*** *node never fails. A* ***faulty*** *node behaves identical to a correct node up to a time t, and then crashes by halting.*

Discussion

Assuming that communication via the network is the only way of interaction in the distributed system, a faulty node sends only messages that a correct node would have sent and, at some time t, completely stops sending further messages. Crash-stop is the simplest fault model. It is, however, a questionable assumption that a correct process never crashes and a crashed one never recovers. A more realistic assumption is made in the crash-recovery model. In this model, processes fail by crashing and later may recover again. On failure, they loose the content of their volatile memory. Algorithms for the crash-recovery model usually make use of *stable storage* that can survive crashes.

Crash-Recovery Failure

Definition 2.6 (Crash-Recovery Failure Model) *In a crash-recovery failure model, a node may be either* ***up*** *or* ***down****. When a node is up, it behaves according to its specification; when it is down, it has stopped executing and does not send any messages. An* ***up*** *node may crash any time, turning into the* ***down*** *state. A* ***down*** *node may recover at any time, turning into the* ***up*** *state. A* ***good*** *node is either one that never crashes, or one that crashes and recovers a finite number of times and, after some time t, is eventually always up. A* ***bad*** *node is one that crashes and recovers infinitely often or that, after some time t', is permanently down.*

Discussion

The crash-recovery model is more realistic than the crash-stop model. Still, both models make the same assumption that a node either behaves strictly according to its specification or is stopped. This assumption may not always be true: Hardware failures may have unpredictable side effects that cause arbitrary behaviour. Even worse, a node might get under control of an attacker, who could force it to perform malicious steps that affect the whole distributed system. The Byzantine failure model covers such kinds of misbehaviour.

Byzantine Failure

Definition 2.7 (Byzantine Failure Model) *In a Byzantine failure model, a correct node behaves according to the specification. A faulty node may take arbitrary steps and send out arbitrary messages to other nodes.*

The Byzantine failure model is the most comprehensive of all. Faulty nodes may show arbitrary behaviour; no assumptions about their behaviour are made. Often, this model is too general in practice, as it permits that faulty nodes do impractical things such as forging the digital signature of other correct nodes.

Restricted Byzantine Failure

Definition 2.8 (Restricted Byzantine Failure Model) *A restricted Byzantine model is identical to a Byzantine model, with the exception that the behaviour of faulty nodes is limited to computationally feasible steps.*

This definition relays on the vague notion of *computationally feasible steps*. For the analysis of specific algorithms, the definition needs to be made more precise. A frequently used clarification is the assumption that faulty nodes may not forge cryptographic signatures of correct nodes. The restricted Byzantine model allows the use of algorithms that function correctly given the unforgeability of digital signatures.

Other Failure Models

This thesis uses only the failure models defined by Definitions 2.5–2.8. Sometimes, other authors use additional failure models; among these models, *timing failures*, *omission failures*, and *value failures* are the most prominent cases. A *timing failure* occurs if a message with the right content arrives late. Considering such failures requires a synchronous system model and is not useful in the case of asynchronous or eventually synchronous models. An *omission failure* occurs if a message is not received at all. Instead of using such a failure model, this thesis assumes a network with *fair lossy channels* (see subsequent definition). *Value failures* are similar to Byzantine failures, but usually are restricted to specific domains. For example, a replication infrastructure might assume that the replicated application produces arbitrarily wrong values (value failures), whereas the replication infrastructure does not.

Failure Model: Variability Required

Ideally, a replication infrastructure should support an arbitrary failure model, selected according to application requirements. If an infrastructure supports only limited variants of the failure model or just one fixed model, it will not provide an optimal solution. A Byzantine failure model is the most general, but usually requires a more complex and costly algorithm. If the application requires tolerating crashes only, an infrastructure for the Byzantine failure model will offer unnecessarily poor performance. Many existing fault-tolerant systems are limited to tolerating crash-stop node failures. This limitation makes them inadequate if other kinds of faults are to be expected. As a result, it is desirable to have an infrastructure whose failure model can be tailored to the requirements of the application.

Network Failures

Fair Lossy Channel

A general approach to modelling communication channels is the fair lossy channel abstraction [BCBT96]. This abstraction is used throughout this thesis.

Definition 2.9 (Fair Lossy Channel) *A fair lossy channel is a communication channel that (a) does not invent any messages (i.e., if P_{rx} receives a message m from P_{tx}, then P_{tx} has sent the message m), (b) does not loose messages infinitely often (i.e., if P_{tx} sends a message m infinitely often to P_{rx}, P_{rx} will receive it infinitely often), and (c) does not duplicate a message infinitely often (i.e., if P_{tx} sends a message m finitely often to P_{rx}, P_{rx} will not receive m infinitely often).*

This definition makes only minimal assumptions about the channel behaviour. A fair lossy channel allows the implementation of quasi-reliable channels by retransmitting messages [BCBT96]. This model implicitly includes the possibility of temporary link failures and temporary network partitions.

Non-Operational Channels

In an asynchronous or eventually synchronous model, it is not feasible to distinguish between operational and non-operational channels. These models do not impose fixed bounds on the message transmission delay. There is no observable difference between a temporarily non-operational channel and one that is just slow.

Failures and Partitions

If, in practice, a communication channel fails or the network partitions, the failure can have one of the following consequences. Either, the system continues to operate in the largest available partition; for most protocols, this strategy requires a quorum of nodes (usually a majority) to be part of this partition. Or, the system blocks until sufficiently many channels become available again.

2.5 Summary

Distributed Object-Oriented Systems

Distributed object-oriented systems apply the principles of object orientation to distributed systems. Services are encapsulated as objects, and remote method invocation is used for interaction. Adaptor code (stubs and skeletons) is automatically generated from interface definitions; clients load the stub code when binding to a remote reference. Popular middleware systems, such as CORBA, Java RMI, and .NET Remoting, provide an infrastructure for systems with such a client-server structure. The CORBA-based Aspectix middleware supports a more general *fragmented-object model*; it allows the use of instance-specific code instead of fixed stubs and provides mechanisms for dynamic quality-aware adaptation.

Replication

The most important domains for replication are file systems, databases, and distributed objects. In all cases, strategies for replica management are required to keep replicas consistent. The focus of this thesis is on active object replication, which uses a consensus-based consistency strategy. For CORBA objects, the OMG offers FT-CORBA as a standard architecture for object replication. FT-CORBA defines a management architecture, generic interfaces, and CORBA references to replica groups, but leaves details such as consistency protocols to individual vendor implementations.

Definitions

The design and evaluation of a replication infrastructure requires an exact definition of the system model. In this thesis, a distributed system is defined as a set of processes that interact by exchanging messages over a computer network. The *synchrony model* imposes bounds on execution speed and message transmission delay. The *failure model* specifies the kind of failures that are expected to happen in the distributed system.

Chapter 3

Existing Object Replication Systems

Overview This chapter examines in detail existing replication support in distributed object middleware. The discussion focuses on middleware integration and development support, thread execution models, and group communication architectures; emphasis is put on the deficiencies that this thesis addresses.

Middleware Integration and Development Support First, we define a set of criteria as a means to evaluate object replication systems. The discussion of high-level aspects of these systems covers the addition of replication mechanisms to the middleware, the development process of replicated objects, and mechanisms for reconfiguration and adaptation.

Thread Execution Models for Replicated Objects After that, the focus is put on *thread execution models* for replicated objects. As multithreading is a potential source of nondeterminism, many existing systems use a simple single-threaded execution model. This approach, however, limits the performance, eliminates useful synchronisation concepts (such as coordination with condition variables), and faces the risk of deadlocks.

Flexible and Reconfigurable Group Communication Finally, *group communication* is discussed as a basic mechanism for maintaining replica consistency. An overview of message ordering properties, reliability guarantees, and variants of implementation strategies provides the necessary background. The support for multiple failure models and dynamic reconfiguration in existing systems are analysed in detail.

Contributions The contributions of this thesis address these three areas. First, replication on the basis of fragmented objects, automated code generation, and the use of semantic annotations allow a simple and flexible development of replicated objects. Second, this thesis defines new strategies for deterministic multithreaded execution. Third, it presents a group communication system that allows variability of the failure model and supports dynamic reconfiguration in an efficient, fault tolerant, and consistent way.

3.1 Middleware Support for Replication

Overview This section analyses the state-of-the-art of replication support in distributed object-based middleware systems. We define the *PECSAR* criteria (portability, efficiency, client transparency, servant transparency, adaptivity, and reconfigurability) as a means to evaluate replication support in middleware systems.

The discussion covers three main issues: first, the middleware integration from a client point of view; second, the development process of replicated objects; and third, the support for adaptation and reconfiguration.

3.1.1 The PECSAR Criteria

PECSAR Criteria

This thesis defines the six *PECSAR* criteria to evaluate a replication infrastructure: **P**ortability, **E**fficiency, **C**lient transparency, **S**ervant transparency, **A**daptivity, and **R**econfigurability.

Portability

PORTABILITY: A replication infrastructure is *portable* if it does not require specific internal modifications to the middleware, but instead can be used as an extension to a standard middleware system. With a portable infrastructure, a fault-tolerant application can be used on any implementation of a specific middleware system. Portability is more difficult to achieve for heterogeneous platforms such as CORBA than for more homogeneous systems such as Java RMI.

Efficiency

EFFICIENCY: The *efficiency* of replication support can be evaluated directly by the latency and throughput of method invocations on a replicated object. The latency is defined by the average total time that a method invocation takes, while the throughput refers to the maximum number of invocations per time interval the replicated object can handle. As the efficiency may differ significantly between executions with and without failures, it is necessary to analyse both cases. The *efficiency without failures* is evaluated in optimal executions without node failures. As non-replicated systems do not handle node failures, it is a proper criterion for a direct comparison between replicated and non-replicated objects. The difference in latency and throughput characterises the overhead of replication and the performance gain that can, for example, be obtained by load balancing. The *efficiency with failures* is an important criterion for comparing fault-tolerance properties of different replication infrastructures. It indicates how well the infrastructure can hide failures from client applications. Ideally, a failure should be masked fully, in which case the latency and throughput are identical with and without node failures.

Client Transparency

CLIENT TRANSPARENCY: The replication infrastructure provides *client transparency* if a client application does not notice any syntactical or semantic difference between accessing a non-replicated and a replicated object. In a transparent system, a non-replicated object instance can be replaced by a replicated one without requiring any modifications to the client. *Full client transparency* means that no intervention at all is needed at the client side. If the client binds to a remote reference, the middleware infrastructure automatically loads the code for accessing the replica group. The decision about loading a different code instead of a standard client stub is triggered internally on the basis of information in the remote reference. *Source-code client transparency* is a weaker form of client transparency. It may be necessary to explicitly inform the local middleware instance about which object is replicated. This may, for example, be done by registering information about replicated objects at system startup. The client source code that is used to invoke remote methods, however, requires no modification.

Servant Transparency

SERVANT TRANSPARENCY: The infrastructures offers *servant transparency* if no modifications have to be made to a non-replicated object implementation (servant) that is going to be replicated. Although this criterion looks similar to client transparency, it is much harder to achieve in practice. On the one hand, the replicated entity must provide additional functionality, such as support for state transfer. On the other hand, the replica implementation is subject to restrictions imposed by the replication infrastructure, such as the requirement of strictly deterministic behaviour.

Adaptivity

ADAPTIVITY: A replication infrastructure is *adaptive* if it provides variants for some task and is able to change the variant automatically without external intervention. Adaptivity can be used for self-configuration and self-optimisation. It requires some kind of observer/controller structure that observes system properties and autonomously requests configuration changes.

Reconfigurability

RECONFIGURABILITY: An infrastructure is *reconfigurable* if it implements variants for some task and allows the variant to be changed manually at runtime. That is, an external entity can request that the configuration of the replication infrastructure be changed. This external entity usually is a (human) administrator that interacts with the infrastructure; it can also be an application that runs on top of the infrastructure.

3.1.2 Adding Fault-Tolerance Support to Object Middleware

Overview

This section discusses the addition of fault-tolerance support to distributed object middleware. Existing systems for object replication are examined in detail regarding the properties of portability, efficiency, and client transparency. The main focus is put on replication support in CORBA-based middleware systems, but most observations similarly apply to non-CORBA systems. CORBA has to support heterogeneity and interoperability between platforms from multiple vendors. In a homogeneous system such as Java RMI, portability and transparency are less difficult to achieve.

Implementation Patterns

Most existing implementation strategies use the interception approach, the service approach, or the integration approach. In addition, some systems employ hybrid approaches that do not exactly fit into a single category.

Interception Approach: Eternal

The *interception approach* was introduced by the Eternal system [MMSN98]. Low-level IIOP messages are intercepted at the interface between the CORBA ORB and the operating system. This approach makes replication fully transparent to client applications. Furthermore, it works with any off-the-shelf ORB without internal modifications; in other words, it supports cross-ORB portability. On the negative side, the interception depends on support from the operating system. In addition, the replication middleware has to analyse low-level byte streams, and semantic information is not available to the replication middleware; this disadvantage introduces a slight performance penalty that other approaches can avoid.

Service Approach: OpenDREAMS/OGS, DOORS

In the *service approach*, the fault-tolerance mechanisms are encapsulated into an application-level service. Clients interact with an object group service to access the replication group. Usually, the service is started locally on each client to avoid that the service represents a single-point-of-failure. Prominent systems that use this approach are OpenDREAMS/OGS [FGS98, Fel98] and DOORS [NGYS00]. Implementing the object group service as an ordinary CORBA service makes it easily portable across different ORB implementations. This approach, however, does not provide client transparency, as the client needs to be aware of the object group service. All invocations to the client require the indirection step through the application-level group service. The indirection adds overhead that decreases efficiency.

Integration Approach: Orbix+Isis, Electra

In the *integration approach*, the middleware ORB is directly augmented with functionality for fault-tolerance support. The most prominent representatives of the integration approach are Orbix+Isis [BR94] and Electra [Maf95]. This approach usually requires that both client and server application run on the same ORB implementation; it does not provide portability. On the positive side, this approach is efficient, as no unnecessary indirections are used. Furthermore, it provides client transparency.

Hybrid Approaches: FTS

A hybrid approach is used by FTS [FH02], which combines CORBA portable interceptors with a custom group object adapter implementation (GOA) to provide fault tolerance in a way that is not compliant to the FT-CORBA standard. The portable interceptor automatically redirects invocations to a group service. The interceptors are not implicitly defined by the IOR, but need to be manually installed at the client. Hence, this approach does not achieve full client transparency, but only source-code client transparency.

Non-CORBA Approaches: AROMA, JGroup, Filterfresh

Replication support can also be added to *non-CORBA middleware infrastructures*. A prominent example is the AROMA system [NMMS00], which transparently enhances the Java RMI system with mechanisms for consistent object replication. It intercepts TCP/IP connections of standard RMI remote invocations at the transport layer and maps them to a reliable, totally ordered group communication protocol. It thus applies the interception approach of Eternal (see above) to Java RMI. As an alternative to such a low-level approach, the Java RMI environment provides extensibility features that allow the developer to use custom mechanisms in place of the internal *stub and skeleton layer* and *remote reference layer*. Jgroup [Mon99] and Filterfresh [BCH$^+$98] both exploit these extensibility features to add replica groups to Java RMI, while maintaining client transparency.

3.1.3 Development of Replicated Objects

Requirements and Restrictions for Developers of Replicated Objects

Implementing a replicated object is more complicated than implementing a non-replicated one. Although replication can be made transparent to clients that access a replicated object, it is hard to achieve servant transparency. The object developer usually has to adhere to strict requirements on replica implementations. On the one hand, he has to implement additional functionality, such as mechanisms for state transfer. On the other hand, restrictions are imposed on the replica implementation. For example, the targets of remote interactions must support duplication suppression or need to be idempotent to maintain invocation semantics. Usually, replica implementations must not contain sources of nondeterminism. Furthermore, if the developer provides additional semantic knowledge about the replicated object, the replication mechanisms can be made more efficient.

State Transfer

Necessity of State Transfer

State transfer is an essential mechanism for replication. It is needed to recover crashed replicas and to create new replicas; the only exceptions are fully stateless objects, which are not further discussed in this thesis.

Manual Approach

In a *manual approach*, the developer of a replicated object has to take direct provisions for state transfer. For example, in a FT-CORBA infrastructure [OMG04] the developer has to implement a `get_state()` and a `set_state()` method in all replicated objects (except for stateless replicated objects). This approach requires the developer to manually provide code for serialising the object state into transfer data and for initialising the object state form this data. Infrastructure support is limited to implicitly invoking the state-transfer methods on creation or recovery of a replica.

Programming Language Approach

In a *programming language approach*, the existing serialisation support of a programming language is used. For example, the Java programming language allows automatic serialisation of classes that are marked with the `Serializable` marker interface [GJSB05]. This approach is fully transparent to the developer, but it is only possible with some programming languages and it does not support state transfer in heterogeneous systems.

Invocation Semantics

A replicated objects aims at providing the same invocation semantics as a non-replicated one (see Section 2.1.1). Failure handling can cause a repetition of invocations. A client that interacts with a single replica of a replication group has to re-issue its invocation after the failure of the contact replica. As it is unknown whether the replica group has received and executed the client invocation, it is essential to detect and suppress a re-execution of the method. A similar kind of re-invocation can happen if the client is a replica group that invokes a remote invocation on another servant. Usually, a single client replica is selected to issue the invocation and later distribute the result to all client replicas. The failure of the interacting client replica makes a repetition of the invocation necessary. The established solution to this problem is to include unique identifiers with all invocation requests. For example, FT-CORBA [OMG04] specifies that such an ID is passed as *invocation context* together with the remote invocation. In a replicated client, it is important that all clients generate the same identifier for the same invocation.

Deterministic Replica Behaviour

Sources of Nondeterminism

Determinism is an important prerequisite for consistent object replication. A reproducible state is obtained in all replicas only if the replica behaviour is deterministic. An object implementation, however, can contain various sources of nondeterminism. In *local system interactions*, an object obtains values from the local system infrastructure (operating system and libraries). For example, it calls API functions of the system to read or write local files from/to disk, to query local values (e.g., system time, CPU load, and hostname), and to generate random numbers. In *external interactions with the environment*, the local instance interacts with external entities. For example, the object implementation might resolve a hostname via the Internet domain name system (DNS). A DNS server can return different IP addresses for the same name. This behaviour is frequently found in practice, e.g., for the purpose of load balancing. *Multithreading* is another source of nondeterminism. If state information of an object is accessed by concurrent threads, mutexes are usually used to serialise the access. In replicas, however, the relative execution speed of threads cannot be predicted, which can result in different orders in which threads access the state. Section 3.2 discusses the multithreading problem in more detail.

Strategies to Handle Nondeterminism

In practice, non-replicated object implementations frequently use nondeterministic operations. The requirement of replica determinism makes it difficult to re-use these implementations for replicated objects. Several strategies exist in practice to cope with this problem [SN06].

Ignoring Nondeterminism

In some instances, the nondeterminism of operations can simply be *ignored.* This approach is trivially possible if the replica implementation makes a nondeterministic operation and ignores the result. More generally, nondeterminism is acceptable if it does not influence the relevant replica state. As an example, a replicated service could pass time stamps in messages that it sends to a client. If the time stamp is not stored in replica state and the client uses one reply only (which it receives from an arbitrary replica), it is irrelevant if other replicas created a different time stamp.

Forbidding Nondeterminism

The simplest approach, used by most existing systems (e.g., FT-CORBA [OMG04]), is to *forbid* any nondeterministic code in replicated objects. If a potentially nondeterministic operation needs to be used, the developer must use a deterministic implementation of this operation. For frequently used operations (such as generating random numbers or time stamps), the replication infrastructure can provide such alternative implementations that guarantee that all replicas will obtain the same value. A simple re-use of existing non-replicated object implementations is not possible in this approach, as the object needs to be adjusted to use the deterministic alternatives.

Runtime Interception

Runtime interception can be used to transparently redirect nondeterministic operations to deterministic variants provided by the replication infrastructure. For example, the Eternal system intercepts library functions at the operating-system level to provide consistent time or mutex locking for multiple threads [NMMS97]. Another approach for interception is to use *code analysis* to replace nondeterministic operation by deterministic variants in the replica code [SN04].

Semantic Knowledge

Transparency and Semantic Information

Transparency is often seen as a desirable feature of replication infrastructures. Full servant transparency implies that a developer can use existing servant implementations without modifications for replicated objects. Such existing implementations do not provide explicit semantic information. Felber et al. [FJRS01] argue that the reliability and dependability of applications can be improved by exploiting semantic knowledge. The authors present their vision of an intelligent middleware which has access to some kind of *semantic repository.*

Obtaining Semantic Knowledge

Semantic information can either be obtained automatically by code analysis or can be provided by the servant developer. Automatically inferring semantics from implementations preserves servant transparency, but is only possible in some situations. For example, code analysis could reveal that a method implementation only reads the object state, obtaining the semantic information that the method is *readonly.* The servant developer usually knows exactly what the properties of his implementation are, and thus he is easily able to explicitly provide such semantic knowledge, even in cases in which automated inferring is not feasible.

Runtime Middleware Support

For the runtime support for semantic knowledge, it is irrelevant how the semantic information is obtained. Both explicit developer annotations and knowledge obtained by automated code analysis can be used to provide the relevant information to the runtime infrastructure. This infrastructure can then adjust its behaviour accordingly.

3.1.4 Dynamic Adaptation and Reconfiguration

Purposes of Adaptivity and Reconfigurability

In the following, the focus is shifted from the static (portability, efficiency, client transparency, and server transparency) to dynamic PECSAR criteria: adaptivity and reconfigurability. These features can be used to optimise system properties such as resource usage, throughput, and latency, while maintaining the desired quality-of-service properties. In addition, adaptivity and reconfigurability allow the system to react to changes in user expectations or environment properties.

Targets of Adaptation and Reconfiguration

Variability in the provided replication mechanisms is the basic prerequisite for any kind of adaptation and reconfiguration. Multiple variants for performing individual tasks have to be provided by the infrastructure. Various tasks can be considered for runtime reconfiguration, and some of them are also suitable for autonomous adaptation. Adaptation usually is implemented by an internal observer/controller component that observes the environment and aims at selecting the optimal variant. Important tasks for reconfiguration are the number of replicas, the replication style, the failure model, consistency protocols, and internal timing parameters.

Number of replicas

The *number of replicas* is usually reconfigurable in any replication infrastructure. The reconfigurability is essential for removing a failed replica from or adding a new replica to a replication group. For example, the FT-CORBA standard defines an *automatic* membership style, in which the replication manager knows the minimal number of required replicas. If the actual number drops below this limit due to failures, the manager uses pre-defined remote factories to create a new replica on an available node. A *dynamic adaptation of the number of replicas* can be found in some existing object-replication systems. The most prominent example is AQuA [RBC$^+$03], a fault-tolerant CORBA infrastructure using the interception approach. AQuA provides an *adaptive* and *quality-of-service–aware* architecture. The minimum number of replicas is fixed, but the actual number is autonomously adapted on the basis of the system load. On high-load situations, additional replicas can be created for the purpose of load balancing. In addition, the type of operations (read-only or modifying) is taken into account, as load balancing improves only read-only operations, whereas modifications become more expensive with an increased number of replicas.

Location of Replicas

The aspect of dynamically configuring the *location and number of replicas* has also been addressed in other contexts such as distributed database systems. Wolfson et al. [WJH97] propose an algorithm for distributed databases that adjusts the replica configuration according to read and write patterns of the client. Bal et al. [BBH$^+$98] analyse similar replica placement strategies for a distributed shared memory system. It is generally understood that such strategies are useful to increase application performance. Recently, in the vision of autonomous computing systems, researchers have taken up the same idea in the context of web services and grid applications [KC03, LNP$^+$03]. In these scenarios, the target is not only to improve the efficiency and throughput, but also to simplify administration. We have developed similar concepts for distributed resource management and service placement in the Aspectix middleware [KHR04].

Adaptive Level of Concurrency in Distributed Databases

In the context of distributed databases, a key focus of adaptability is on the *multiprogramming level*, i.e., the number of parallel transactions. Similar to concurrent request execution, the execution of multiple transactions in parallel can increase efficiency. Too much parallelism, however, decreases efficiency, as it causes context switches and increases the probability of conflicts between transactions. For example, Heiss and Wagner [HW91] as well as Mönkeberg and Weikum [MW92] discuss the problem of dynamically selecting an optimal multiprogramming level. Milan-Franco et al. present an adaptive middleware for data replication that examines the system load and the type of requests to perform both local adjustment of the multiprogramming level and global load balancing.

Replication Style

Replication platforms often support *variability in the replication style*. In general, the infrastructure offers mechanisms for both active and passive replication. The selection of a replication style is, in general, only supported at the creation time of a replicated object. A reconfiguration of the replication style could be made by shutting down and reinstantiating the replicated object; this approach, however, results in a period of unavailability. An approach for *dynamically changing the replication style* has been proposed by Felber et al. [FDES99]. This work enables the selection of the replication style with fine granularity on a per-invocation basis. This approach allows, for example, the execution of methods that are nondeterministic or that cause extensive computations with a passive replication style, while other invocations can be executed with an active replication style.

Failure Model

The status quo of existing object replication systems is to assume a *fixed failure model*. Most systems assume a fixed fail-stop or fail-recovery system model; intrusion-tolerant systems, such as the system proposed by Sames et al. [SMN$^+$02], are based on a Byzantine failure model. In general, such existing replication infrastructures do not support multiple variants and cannot offer any adaptation or reconfiguration regarding the failure model. AQuA [RBC$^+$03] supports an intermediate system model, which assumes fail-stop failures of replicas but in addition handles omission failures and value failures. The AQuA infrastructure contains a voter component that is assumed not to fail maliciously; this means that the infrastructure does not tolerate arbitrary Byzantine failures. Furthermore, in the Immune system [NKMMS99] the AQuA architecture was combined with SecureRing [KMMS98], a Byzantine fault-tolerant group communication system. This means that AQuA can be considered as a system that supports variability in terms of failure model. Such an architecture can support reconfigurability, but only at the cost of completely replacing the group communication system.

Consensus Protocols and Adjustable Parameters

At the internal consistency level, variability can be provided by multiple consistency protocols and by adjustable parameters. Whereas adjusting parameters (e.g., failure detection timeouts) is usually possible at any time, the replacement of protocols requires coordination with the execution of requests. A more detailed discussion of existing systems that provide variability at this level is given in Section 3.3.

Low-Level Communication

Low-level communication has a high degree of *variability*. For example, nodes may communicate with hardware multicast facilities, with unreliable datagrams (such as UDP messages), with reliable channels (such as TCP connections), with encrypted channels (such as TLS connections), and with higher-level facility (such as over HTTP connections). It is state of the art that systems allow the selection from a set of communication mechanisms. For example, the JGroups group communication system [Ban98] support TCP connections as well as UDP unicast and UDP multicast. In such systems, the selection of a specific variant is done at startup, and thus it offers no runtime reconfiguration and adaptation. The *lack of reconfigurability and adaptivity*, however, is a serious deficiency of existing systems. Reconfigurability mainly becomes necessary if an application administrator changes communication policies; such change might, for example, demand encrypted communication instead of simple TCP channels. Autonomous adaptation of the communication is important for self-organising and mobile systems. The availability of communication mechanism and their performance depends on the network environment; for example, hardware multicast facilities often are available only in local-area networks. In a self-organising system, the infrastructure might relocate existing replicas or place new replicas on some dynamically selected nodes. After relocation, an autonomous selection of the best low-level communication mechanism is desirable. In a mobile system, a client node might change its network connectivity at runtime, requiring dynamic adjustment of low-level communication.

3.2 Thread Execution Models for Replicated Objects

Overview

Many distributed object replication systems do not allow multithreading in replicated objects. Method invocation requests from clients are executed in a strictly sequential order. This approach avoids any nondeterminism that can arise from thread scheduling. On the other hand, such a single-threaded execution model has serious problems: it lacks performance, it complicates the reuse of existing servant implementations (as these implementations might use mechanisms, such as condition variables, that require multithreading), and it is inherently deadlock-prone. This chapter offers a classification of these problems, discusses existing concepts for multithreaded execution of replica methods, and evaluates their capability to solve the aforementioned problems.

3.2.1 Problems Related to the Thread-Execution Model

Classification of Problems

Depending on the thread execution model, a servant implementation may face the following problems:

Deadlocks Caused by Circular Nested Invocations

CircularDeadlock: Circular nested invocations can cause a deadlock. For example, let's assume that a replicated object A, while executing a method m_{A1}, invokes a remote method m_B at object B, and method m_B in turn invokes a method m_{A2} at object A. If object A operates in a strictly sequential fashion, it will not handle the method invocation of m_{A2} before m_{A1} returns. On the other hand, m_{A1} will return only after m_B (and, consequently, m_{A2}) has returned; this results in a deadlock (see Figure 3.1a).

Deadlocks Caused by Mutual Nested Invocations

MutualDeadlock: Mutual invocations can cause a deadlock. For example, let's assume that a replicated object A executes a method m_{A1}, which invokes a remote method m_{BA} on object B. In parallel, object B executes a

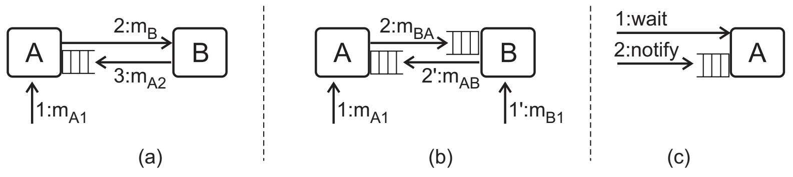


Figure 3.1: Deadlock situations without multithreading

method m_{B1}, which invokes a remote method m_{AB} at object A. Both the invocations of m_{AB} and m_{BA} cannot be handled in a strictly sequential execution, as both objects will execute these invocation only after m_{A1} and m_{B1}, respectively, will have returned. Again, a deadlock is reached (see Figure 3.1b).

Lack of Support for Condition Variables

NoConditionWait: Condition variables are a popular programming concept in multithreaded software that allows a thread holding a mutex m to atomically release the mutex and suspend. Another thread can acquire the mutex m and notify the suspended thread to resume. The suspended thread resumes after it has successfully re-acquired the mutex m. The use of condition variables is possible only in a multi-threaded execution model. The thread waiting on a condition variable needs to be notified by a second thread (see Figure 3.1c). If an implementation that uses condition variables is executed in a single-threaded way, it deadlocks. Thus, without multithreading, workarounds such as periodic polling or the installation of a callback object have to be used, which usually have higher overhead than simply waiting on a condition variable.

Idle Time During Nested Invocations

NestedIdling: Let's assume that a replicated object A executes a method m_{A1}, which invokes a remote method m_B on object B. In a single-threaded execution model, object A will not process any other request before m_B returns and m_{A1} finishes (see Figure 3.2a), but instead will remain idle until the reply is received. This behaviour is less efficient than running a second thread to handle the invocation of a method m_{A2} while the first thread waits for the reply from the nested invocation. Avoiding NestedIdling does not automatically imply that multiple threads run concurrently. In fact, it is sufficient to have one active thread at a time to efficiently use the idle time for processing additional requests (see Figure 3.2b).

Lack of Parallelism on Multi-CPU/Multi-Core CPU Nodes

NoParallelism: Modern computer architectures often include multiple CPUs, allowing true parallel execution of requests. In such an environment, the truly parallel execution of requests leads to better CPU utilisation and consequently increases efficiency (see Figure 3.2c). If the infrastructure does not permit such a concurrent execution of multiple active threads within one object, it provides less efficiency.

Necessity of Explicit Synchronisation of State Access

ExplicitSync: If multiple threads may execute concurrently, access to the shared object state has to be coordinated. The developer has to provide explicit synchronisation statements (such as mutex locks), which serialise concurrent modifications. This problem does not exist if the execution model provides implicit synchronisation, as is the case in single-threaded execution. Concurrent access to the shared state is the only problem related to the thread execution model that is easier to solve without multithreading.

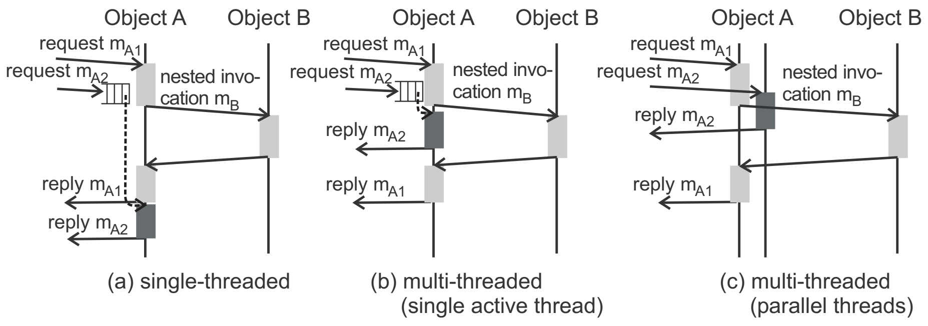


Figure 3.2: Performance comparison of single-threaded and multi-threaded execution

3.2.2 Variants of the Execution Model

Many existing replication systems use a strictly sequential execution model, but some approaches to multithreading that partially solve the preceding problems have previously been published. Existing thread-scheduling strategies can be classified into the following categories:

Strictly Sequential Execution

SEQUENTIAL: In a strictly sequential execution, a request is processed only after the preceding request has been completed. This model is widely used in fault-tolerant middleware systems (e.g. OGS [FGS98], GroupPac [FMLF97]). Given a total order of all incoming requests and deterministic replica behaviour, consistency is easily obtained. The execution model provides implicit synchronisation, removing the necessity to explicitly synchronise state access.

Execution with a Single Logical Thread

SINGLELOGICALTHREAD (SLT): In this model, a single logical thread of execution exists. In a chain of nested invocations, the logical thread may call methods of the same object multiple times. For example, in an interaction pattern as shown in Figure 3.1a, the thread that executes m_{A1} at object A starts a chain of nested invocations that ultimately calls method m_{A2} at object A. In the SLT model, the object A can detect that the invocation m_{A2} belongs to the same logical thread as m_{A1}, permitting the execution of m_{A2}. This way, the CIRCULARDEADLOCK problem is removed. Technically, context information that identifies the originating logical thread is propagated through remote call chains. If an object detects that an incoming request belongs to the current logical thread of executions, it can execute the request with an additional physical thread. No nondeterminism can arise, as the first thread remains blocked in all replicas during the execution of the nested invocation, and resumes only after the additional physical thread has finished. This model was first used in the Eternal system [NMMS99].

Execution with a Single Active Thread

SINGLEACTIVETHREAD (SAT): In this model, multiple physical threads may exist within a replica, with only one of them being active at a time, and all others being blocked (e.g., waiting for a lock or for the return from a nested invocation). Consistency is obtained by a deterministic selection of the active thread. A running active thread is not preempted; if the active thread blocks or terminates, a deterministic strategy is required to resume one of the existing threads or to create a new active thread for handling the next request. If the strategy guarantees that the same choice is made in all replicas, consistency is

	SEQUENTIAL	SLT	SAT	MAT
CIRCULARDEADLOCK	×			
MUTUALDEADLOCK	×	×		
NOCONDITIONWAIT	×	×		
NESTEDIDLING	×	×		
NOPARALLELISM	×	×	×	
EXPLICITSYNC			×	×

Figure 3.3: Taxonomy of execution models and solved problems

maintained. An algorithm using this model was first suggested by Jimenez-Peris et al. [JPPMA00] for a transactional, conversational client-server interaction model. Zhao et al. [ZMMS05] proposed a similar model for RPC-based replicated objects.

Execution with a Multiple Active Threads

MULTIPLEACTIVETHREADS (MAT): In this model, multiple threads may concurrently be active. To maintain consistency, all access to shared data structure needs to be made in a consistent order. The only previously published algorithms in this model are the *Loose Synchronization Algorithm (LSA)* [BWKI02] and the *Preemptive Deterministic Scheduling (PDS)* algorithm [BKI03]. LSA uses a leader-follower model and offers a high degree of parallelism. As a drawback, it adds communication for synchronisation and has disadvantages if the leader fails. PDS is completely free of communication, but it makes strict assumptions on the creation of threads and the acquisition of locks of all existing threads within sequential rounds. Both algorithms assume that simple locks are the only means of synchronisation.

Comparison

Figure 3.3 shows to what extent the various execution models are able to solve the aforementioned problems. Explicit synchronisation of access to shared state is not necessary for algorithms in the SEQUENTIAL and SLT categories. The only benefit from the SLT category, compared to SEQUENTIAL, is that it avoids the CIRCULARDEADLOCK problem. On the other hand, SAT solves all the problems except NOPARALLELISM and EXPLICITSYNC. MAT is the only model that enables true parallelism, which makes it possible to use all computational resources on a multi-CPU hardware or multi-core CPUs.

Low-Level Approaches to Deterministic Multithreading

The nondeterminism that is caused by multithreading can be addressed at several system levels. Some existing research projects use a modified Java virtual machine to implement deterministic replication. Napper et al. (based on a modified Sun JDK 1.2) [NAV03] and Friedman and Kama (based on a modified JikesRVM) [FK03] provide examples for this approach. Other systems ensure determinism at an even lower system level. For example, MARS [KDK$^+$89] is strictly time-driven and periodic at the hardware level, which makes all functional and timing behaviour deterministic. The features of such a platform can be used for deterministic replication [PBWB00]. All these systems support the MULTIPLEACTIVETHREADS category. They all require specifically designed hardware, operating systems, or Java virtual machines to achieve determinism. In contrast, this thesis uses means to enforce determinism of multithreaded replicated services purely at the middleware level, without requiring special low-level support in operating system or virtual machine.

3.3 Group Communication

Group Communication

The term *group communication* in general denotes any form of reliable one-to-many communication (i.e., multicast). Depending on the message ordering properties the following specific variants have been defined by Birman [Bir97]:

- **bcast**: Reliable multicast to all group members, without message ordering guarantees.
- **fbcast**: FIFO-ordered reliable multicast to all group members. If one sender sends multiple messages to the group, these messages will arrive in sender order. No order is guaranteed for messages originating from different senders.
- **cbcast**: Causally ordered reliable multicast to all group members. If one group message m_2 logically depends on m_1, it is guaranteed that m_1 will be delivered before m_2 at all group members.
- **abcast**: Atomic reliable multicast to all group members. All group members receive messages in identical order. Although it is theoretically possible to implement **abcast** without FIFO ordering, it is usually understood to include the **fbcast** properties.
- **cabcast**: Atomic and causally ordered reliable multicast to all group members; i.e., the properties of both **cbcast** and **abcast** are provided.

3.3.1 Implementation Variants of abcast

Overview

Totally ordered group communication is a synonym for abcast and can be implemented in several ways. The survey of Défago et al. [DSU04] gives the most complete overview of totally ordered group communication systems. Implementation approaches can be classified into three different categories. First, *sequencer-based* systems use a designated node (the "sequencer") to specify the message order. Second, *sender-order* systems define the order of messages directly at the sender side. Third, *consensus-based* group communication systems use fault-tolerant consensus algorithms to obtain a consistent messages order.

Sequencer-based Group Communication

Sequencer-based group communication can be implemented in various ways. Usually, one fixed group member is elected as sequencer; if this node fails, a new sequencer needs to be found. Let P_i be the node that wants to send a message m to all group members in total order. The sequencer needs to define a consecutive sequence number $seq(m)$ for each message m. The interaction of P_i, the sequencer, and the group is defined by a sequence of unicast (U) and broadcast (B) operations, which can happen in one of the following ways.

- *UUB Model* UUB model: P_i unicasts a control message to the sequencer, which in turn unicasts back a sequence number; finally, the node P_i broadcasts m together with $seq(m)$ to all group members.
- *UB Model* UB model: P_i sends its message m to the sequencer, which in turn broadcasts m and the sequence number $seq(m)$ to all group members.

BB Model

- BB model: P_i broadcasts m to all group members (one of them being the sequencer). After the reception of m, the sequencer defines the sequence number of m and broadcasts this value together with a unique message ID of m. Group nodes process a message only after having received its sequence number.

Comparison of UUB, UB, and BB Model

None of the three methods is definitely superior to all others. The UB model has minimal latency and minimal message cost. The disadvantage of this model is that it causes the highest load on the sequencer, which has to handle all broadcast operations itself. The BB model aims at reducing this overhead. The sequencer no longer has to broadcast application messages, which is a benefit particularly if the message size is large; in addition, it may define the message order of multiple application messages with a single broadcast. The UUB model fully decentralises the broadcast operation, but it adds one round-trip delay to the message transmission, which increases message latency.

Moving Sequencer

Algorithms such as the one proposed by Chang and Maxemchuck [CM84] use a moving sequencer mechanism for load balancing to avoid that the sequencer becomes a bottle-neck. Instead of having a fixed sequencer, the sequencer role changes periodically. This approach is best combined with the BB model to avoid that senders need to keep track of the sequencer. This approach, however, involves coordination overhead for changing the sequencer and leads to more complex protocols.

Sender-Order Group Communication

In *sender-order group communication*, the order of messages is defined by the message senders. Such a strategy is easily implemented if only one sender at a time is permitted. Otherwise, if multiple senders emit messages concurrently, they can assign logical time stamps to the messages, which define a global total order.

The single-sender variant of sender order is usually implemented with a token-based approach. Only the token owner is allowed to send group messages and to assign sequence numbers to messages. Some strategy to pass the token around is used. On passing the token, the old token owner informs the next node about the highest used sequence number. This simple approach does not allow that multiple senders concurrently send messages to the group.

The alternative is to allow concurrent message transmission, but to include logical time stamps in the messages. For example, Lamport's logical clocks or vector clocks can be used for this purpose. A node delivers a message as soon it is sure that no other message with a smaller time stamp can arrive.

Consensus-based Group Communication

Using *consensus algorithms* for totally ordering group messages was first proposed by Chandra and Toueg [CT96] and subsequently used by several existing systems. For example, Rodrigues and Raynal [RR00] apply the Chandra-Toueg transformation—which assumes a crash-stop fault model—to the crash-recovery model. Mostefaoui and Raynal [MR00] describe an optimisation that restricts the use of the consensus algorithm to situations where asynchrony and crashes prevent nodes from obtaining a simple agreement on message order. Consensus-based group communication is also used in the Aspectix group communication system this thesis will present in Chapter 6.

Uniform and Non-Uniform Group Communication

Uniform and Non-Uniform Group Communication

Group communication variants can differ not only in their ordering properties, but also in the reliability guarantees they offer in the case that nodes fail during message transmission. A node P_i *delivers* a message m if the group communication system on that node passes this message to the application.

Definition 3.1 (Non-uniform reliable group communication)
Non-uniform multicast guarantees that if ***a non-faulty*** *group member P_i delivers a message m, all non-faulty group members will eventually deliver m.*

Definition 3.2 (Uniform reliable group communication) *Uniform multicast guarantees that if* ***any*** *group member P_i delivers a message m, all non-faulty group members will eventually deliver m.*

Relevance of Difference

The difference between uniform and non-uniform group communication is important if a node P_i delivers a message m and subsequently crashes. With non-uniform semantics, the surviving nodes may decide to deliver a different message m'. As long as the behaviour of the crashed node is irrelevant, non-uniform semantics is sufficient. Otherwise, if the delivery of m at P_i has caused side effects (e.g., external interactions or response messages), these side effects will be inconsistent with the state of the nodes that did not crash. The consequences that this aspect has on object-replication are discussed in more detail in Section 4.2.

3.3.2 Failure Models, Reconfiguration, and Adaptation

Failure Models in Group Communication Systems

Group communication systems exist for various failure models. For example, systems such as Ensemble [Hay98], xAMp [RV92], Spread [ADS00], and JGroups [Ban98] assume a fixed crash-stop or crash-recovery failure model. Byzantine fault tolerance is found in intrusion-tolerant group communication systems such as SecureRing [KMMS98] and RamPart [Rei94].

Variability of the Failure Model

Variability of the failure model is not considered in existing systems. If a replication infrastructure wants to offer configurable failure models with such group communication systems, the only option is to use a different group communication system for each model. While this is a feasible option at system startup, changing the group communication system for reconfiguration at runtime would be an expensive operation. It takes a significant amount of time to shut down the old system and start the new system. Furthermore, such a replacement requires coordination with all access to the group communication system, and it causes intensive resource use (e.g., for loading and initialising the library of the new communication system). In practice, existing systems usually do not support such reconfiguration.

Runtime Reconfiguration

The static nature of existing group communication systems manifests itself not only in the failure model, but also in other configurable elements. For example, the JGroups system [Ban98] has a modular structure that offers a high degree of configurability. The configuration of the modular protocol stack, however, is determined at the creation time of a communication group. Once again, the only way for runtime configuration is a complete re-creation of the configuration group.

Adaptive Group Communication Systems

Many configurable parameters of existing group communication systems cannot be changed easily at runtime, which also excludes autonomous adaptation of these parameters. Some existing works, however, discuss the adaptation of single internal parameters such as retransmission or failure-detection timeouts. For example, Bertier et al. [BMS02] present a failure-detector implementation that observes communication to estimate future arrival times to minimise the failure detection time while avoiding false detections. Such adaptive mechanisms help to increase the performance of group communication systems, but they do not consider large changes such as using a completely new message ordering algorithm.

3.4 Contributions of this Thesis

The preceding discussion of existing object replication system allows the identification of several deficiencies. This thesis presents the FT*flex* architecture for fault-tolerant object replication in distributed systems, which addresses several of those deficiencies. The contributions of the FT*flex* architecture concern (a) the middleware integration and development support, (b) deterministic multithreading in replicated objects, and (c) flexible and reconfigurable group communication.

3.4.1 Middleware Integration and Development Support

Fragmented Objects

FT*flex* uses fragmented objects to provide fault-tolerant replication in a CORBA-based architecture. It supports active replication with flexible failure models, ranging from crash-stop to Byzantine failures. Chapter 4 describes the FT*flex* architecture in detail. The FT*flex* approach offers advantages compared to other existing middleware infrastructures for object replication. Our PECSAR criteria can be used to evaluate these advantages:

Portability, Efficiency, and Client-Side Transparency

At the client side, the objective is to combine *portability* with *efficiency* and *transparency*. Portability requires the client-side middleware to load code for accessing replica groups transparently, without having a priori provisions for replication in the middleware. FT*flex* uses the Aspectix middleware, which enables such code loading with the fragmented-object model. Remote references to fragmented objects directly specify the instance-specific code that is to be used for accessing replica groups. This results in a mechanism that is portable across all platforms that can handle references to fragmented objects. Furthermore, the FT*flex* approach offers full client transparency. As the code specified in the reference is loaded directly, the approach avoids any unnecessary indirections that occur in interception-based systems.

Servant Transparency: Semantic Information and Automated Code Generation

The FT*flex* system does not aim at providing full *servant transparency*. Instead, it offers the developer the possibility to directly influence the replication mechanisms with semantic annotations. These annotations can be used to improve the efficiency by optimising the execution strategy of remote invocations and by placing functional code directly at the client side. In addition, FT*flex* advocates code generation and transformation for simplifying the development of replicated objects. Thus, the modifications that need to be made to existing servant code for enabling replication are minimised. As a result, the FT*flex* infrastructure offers better flexibility for developers and allows better optimisation of replication strategies on the basis of semantic knowledge.

Adaptivity and Reconfigurability

Dynamic adaptation and reconfiguration are another important topic of this thesis. The FT*flex* prototype uses only active replication, without considering variability in the replication style, as this issue has previously been discussed in detail by Felber et al. [FDES99]. Adaptation and reconfiguration is mainly provided at the level of group communication, which is used for low-level consistency management. At the fragmented-object level, an interface of the Aspectix middleware enables client applications to pass policies to the fragmented object, which can trigger an internal reconfiguration. The broader problem of distributed management of services and resources, which includes the adaptation of the number of replicas and the placement and migration of replicas, is also addressed in the Aspectix project [KHR04, KRH05], but is outside the scope of this thesis.

3.4.2 Deterministic Multithreading in Replicated Objects

Enabling Multithreading

A single-threaded execution model is found in many existing replication infrastructures, but it causes serious problems for many applications. The possibility that existing non-replicated code depends on multithreading (for example, because it uses condition variables) would complicate the reuse of that code for replicated objects. For this reason, this thesis proposes new strategies that enable multithreaded execution in combination with active replication of objects. This topic is covered in detail in Chapter 5.

Full Java Synchronisation

In previously existing solutions for multithreading, the synchronisation model is limited to binary mutex locks. In contrast, the FT*flex* infrastructure offers a synchronisation model that includes reentrant mutex, condition variables, and time bounds on blocking wait operations. This model is comprehensive enough to support all core synchronisation mechanisms of the Java programming language, which simplifies the reuse of existing object implementations.

Source-Code Analysis and Transformation

Obtaining determinism in multithreaded objects requires that the middleware intercept all synchronisation statements in the object implementation. This thesis proposes source-code analysis and transformation as a novel approach for such interception. The intercepted operations are forwarded to an instance of the Aspectix Deterministic Thread Scheduler (ADETS).

Deterministic Scheduler Module

This thesis presents four variants of the ADETS scheduler. ADETS-SAT provides a solution for the SingleActiveThread model and includes support for native Java synchronisation (i.e., reentrant locks, condition variables, and time bounds on wait operations). ADETS-LSA and ADETS-PDS implement the LSA and PDS algorithms of Basile et al. [BWKI02, BKI03], and extend them for the synchronisation model of the FT*flex* infrastructure. Finally, this thesis specifies ADETS-MAT, a novel scheduling algorithm in the Multiple-ActiveThreads category. Unlike the LSA algorithm, ADETS-MAT does not require communication for granting locks. Unlike the PDS algorithm, it has no restrictions on the creation of threads by external requests or the number and frequency in which threads acquire locks.

3.4.3 Flexible and Reconfigurable Group Communication

Aspectix Group Communication System

The contributions of FT*flex* at the level of group communication and consistency management focus on the support for variable failure models, runtime reconfiguration, and autonomous adaptation. Flexible and reconfigurable replica consistency strategies are encapsulated within the Aspectix group communication system (AGC). This system provides a consensus-based, totally ordered group communication mechanism, and is described in detail in Chapter 6.

Variable Failure Models

The first objective of the Aspectix group communication system is to support a wide range of failure models encapsulated within a homogeneous system. The AGC uses implementations of well-known consensus algorithms for various failure models to obtain total message ordering. The supported failure models include crash-stop, crash-recovery, and Byzantine failures.

Algorithmic Variability

The realisation of the message-ordering strategy within a pluggable consensus module not only permits multiple failure models, but also allows the use of algorithmic variants for each failure model. Such variants can support different synchrony models or can differ in metrics such as latency, best-case versus worst-case overhead, and number of messages per consensus decision. By allowing the selection of a specific variant, FT*flex* permits the developer to tailor the group communication system to environment and interaction patterns.

Adaptivity and Reconfigurability

Another important objective of the Aspectix group communication system is to allow dynamic reconfiguration. This thesis defines an extended version of the Chandra–Toueg algorithm for totally-ordered multicast on the basis of distributed consensus. The extended version enables variability of the consensus algorithm and the group membership. On this basis, the AGC supports efficient and consistent run-time reconfiguration. Unlike other systems, the AGC allows that policies can be changed at runtime while maintaining full system consistency, that these reconfigurations have no or only minimal impact on the running system, and that the reconfigurations are fault tolerant themselves. This allows an administrator to dynamically change the configuration. Furthermore, FT*flex* supports autonomous observer/controller components that analyse properties such as network load in order to automatically trigger reconfigurations on the basis of simple rules.

3.5 Summary

PECSAR Criteria

Existing object replication systems use various approaches for adding replication support to middleware infrastructures. The PECSAR criteria (portability, efficiency, client transparency, servant transparency, adaptivity, and reconfigurability) allow an evaluation of such existing infrastructures.

Middleware Integration

Many replication systems provide static infrastructures that offer only little flexibility. In Chapter 4, this thesis proposes the use of fragmented objects and semantic annotations as a way to increase flexibility and efficiency of object replication.

Multithreading in Replicated Objects

The thread execution model is an important aspect of an object replication system. A single-threaded execution, which is used my most existing systems, causes many problems, such as the risk of deadlocks and poor performance. Chapter 5 presents new algorithms for deterministic thread scheduling and proposes the use of source-code transformation for intercepting synchronisation.

Flexible and Reconfigurable Group Communication

Middleware infrastructures usually implement active replication on the basis of totally ordered group communication. Existing systems usually face two limitations: they are restricted to a single failure model, and they lack reconfigurability. Chapter 6 proposes a new group communication architecture, which is implemented in the Aspectix group communication system (AGC).

Chapter 4

Middleware Integration and Development Support

Overview

This chapter presents the fundamental FT*flex* architecture [RKDH06]. FT*flex* adds support for fault-tolerant replication to the CORBA-based Aspectix middleware. The first section describes the use of the fragmented-object model for replicating objects. This approach establishes the basis for the subsequent contributions of this thesis. The next section discusses in detail consistency challenges relating to failure model and group communication semantics and the corresponding solutions in FT*flex*. Furthermore, the mechanisms for supporting state transfer in the replication architecture are explained. After that, semantic annotations are introduced for enabling the developer to tailor the middleware infrastructure according to his needs. Moreover, this chapter discusses replica group management, which is used for replica creation and runtime reconfiguration. Finally, the benefits FT*flex* approach are evaluated.

4.1 Replication with Fragmented Objects

Replication Support with Fragmented Objects

The Aspectix middleware provides a basic infrastructure for fragmented objects (see Section 2.1.3). This infrastructure is used by the FT*flex* replication architecture. FT*flex* directly supports loading object-specific fragment code and offers transparency for clients. The client-side strategy for interacting with the replica group and the replica-side strategy for consistency and group management is implemented within fragments of a replicated object.

4.1.1 Development Process

Overview of the Development Process

The fragmented-object model of Aspectix allows the developer to create arbitrary fragment implementations. The FT*flex* system supports the development of fragments of replicated objects by automatically creating major parts of these fragments. Figure 4.1 illustrates the development process, which consists of defining the global object interface in CORBA IDL, optionally specifying semantic annotations, automatically creating fragment code, implementing the functional parts of the object, and applying code transformations to the fragment code before deployment.

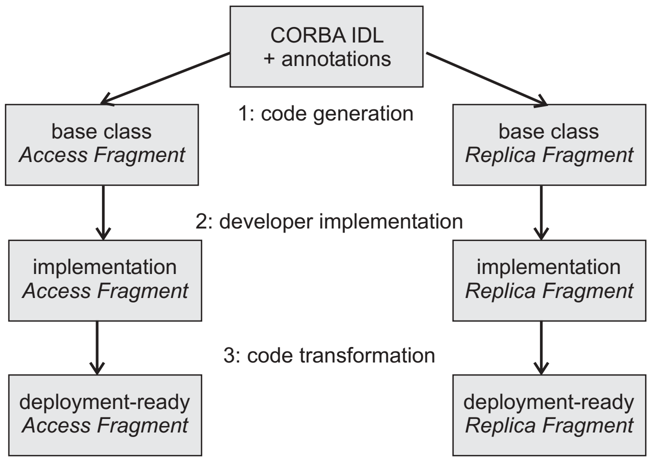


Figure 4.1: Development process of replicated objects with FT*flex*

Specification of Interface and Annotations

The IDL specification of a replicated object is identical to that of a standard CORBA object. This has the benefit that an existing client application that has been developed on the basis of a standard interface definition can be reused without modification to access a replicated object. In addition, FT*flex* enables the developer to influence the IDL-based code generation process using his semantic knowledge about the replicated object. For this purpose, the developer can add annotations to the IDL interface specification. These annotations only influence the internal code generation process, but are irrelevant for client applications.

Automated Code Generation

Both the IDL specification and semantic annotations are used for the automated generation of code. The fragment architecture consists of two types of fragments, *access fragments* and *replica fragments*, as shown in Figure 4.2. The replica fragments contain code for consistency management, while the access fragments are used by clients to access the replica group. For each fragment type, the FT*flex* code generation tool creates a base class. Hence, the transition from an existing implementation to a replicated one is automated as much as possible, with only minimal developer intervention required.

Object-Specific Replication Code

The automated generation of code is an essential part of the presented architecture. First of all, such a generation process is important for the efficient use of the fragmented-object model. If the developer had to implement the fragment code completely manually, the required effort would make the approach hardly acceptable. Besides, considering developer annotations in the generation process enables a flexible customisation and optimisation of the replication mechanisms on a per-object basis.

IDLflex-based Code-Generation Tool

The current prototype of the code generation tool is based on IDL*flex*, an IDL-compiler that generates customisable code [RSH01]. IDL*flex* parses CORBA IDL, evaluates an XML-based mapping specification, and uses this specification to create arbitrary output code. It includes two standard mapping specifications for the Java programming language, one for standard CORBA and one for generic Aspectix fragmented objects. FT*flex* provides a specialisation of the Aspectix mapping specification that defines the creation of access fragments and replica fragments on the basis of IDL interface and semantic annotations.

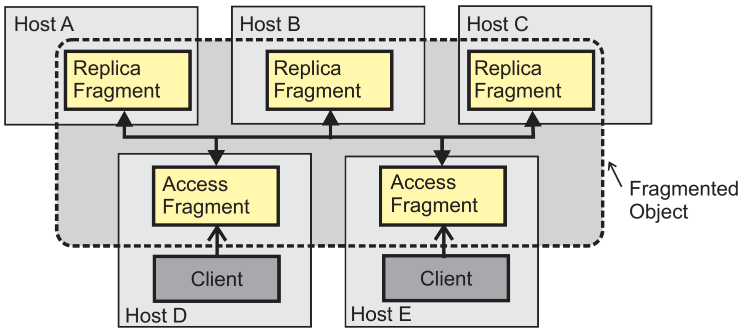


Figure 4.2: Architectural overview of a replicated object in FT*flex*

Functional Implementation

Next, the developer can individually add custom code to the generated basic fragments. The functional implementation of the servant is, in general, implemented in the replica fragments. The developer, however, may also choose to implement some functionality directly in the client-side access fragment. Section 4.4 discusses how the developer can use annotations to adjust the automatically generated code depending on where object functionality is provided.

Code Transformation

As a third step of the fragment development process, a code transformation tool can modify the functional implementation that the developer provides. This approach is used to intercept native Java synchronisation code. Synchronisation operations need to be intercepted and forwarded to a middleware plug-in module that is responsible for deterministic thread scheduling (see Chapter 5). The replacement of all relevant statements with custom code enables such an interception without internal modification to JVM or operating system. The same approach could be used to intercept Java API calls to nondeterministic methods, such as the generation of time stamps or random numbers.

Deployment

Finally, after compilation and deployment, the replicas of the object can be started. FT*flex* provides a simple management infrastructure that enables clients to create replicas on remote locations using a factory approach (see Section 4.5). The factory is used to instantiate replica fragments and to join them to the existing group of replicas. FT*flex* creates an IOR for the replicated object, which can be used by clients for binding to and accessing the replica group. This IOR contains an Aspectix profile that specifies the corresponding access fragment as the initial fragment type that has to be loaded by the client-side middleware infrastructure.

Policy Interface

The Aspectix architecture provides a policy interface to configure the internal structure and to request dynamic reconfigurations. Configuration can either be supported internally within the fragment code, or it can trigger a consistent replacement of the whole local fragment.

Evaluation

The development process of replicated objects using FT*flex* is more complex than the development of a traditional CORBA servant. The main benefits from the FT*flex* approach are the customisation of code generation via a specialised mapping, the ability to use custom developer code at the client side, and the use of code transformation for transparently adding mechanisms such as the interception of synchronisation statements in the replica code.

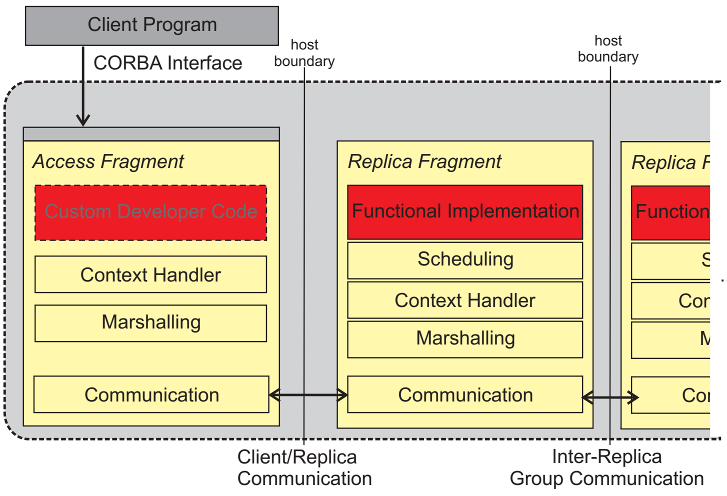


Figure 4.3: Internal architecture of replica fragments and access fragments

4.1.2 Internal Fragment Architecture

The internal structure of access and replica fragments is shown in Figure 4.3. The fragments are internally divided into several components.

Client Interface

A client application accesses the fault-tolerant fragmented object via its IDL-defined interface, as it would access any other CORBA object. If a client binds to the remote reference of a replicated object, the fragmented-object middleware automatically loads a local access fragment. All invocations on the object interface go to the local access fragment.

Clients on Replica Nodes

Only access fragments handle client interactions. If a client is located at the same node as a replica fragment, it also uses a local access fragment to access the replicas. Essential functionality, such as the assignment of unique invocation IDs, is implemented in the access fragment. A direct access from client code to the replica fragment would require the duplication of this functionality in the replica fragments. Such duplication was avoided in the FT*flex* architecture. Instead, an access fragment is loaded in parallel to the replica fragment.

Developer Code in the Access Fragment

As shown in Figure 4.3, the access fragments may contain optional client-side developer code. All client invocations are first passed to this developer code. This allows the provision of object functionality directly at the client side or the implementation of mechanisms such as client-side caching. For all methods that have no custom developer code in the access fragments, the default behaviour is to pass all invocations directly to the generated code that handles the remote invocation at the replica fragments.

Functional Implementation

At the replica fragments, developer code provides the actual object implementation. All remote invocations from clients are finally passed to this object implementation. The functional implementation is the only part of the fragment code that is not automatically created. The code transformation process can, however, manipulate the custom developer code. Such a code manipulation can be used for intercepting nondeterministic operations. For example, the transformation enables a transparent interception of Java synchronisation statements, which is needed for the deterministic multithreading support discussed in Chapter 5.

Scheduling

The *Scheduling* component of the replica fragment is responsible for the internal message management and interacts with a deterministic multithreading plug-in for thread creation; the details of this plug-in are explained in Chapter 5.

Context Handler

The *Context Handler* is responsible for adding unique identifiers to requests. The identifiers consist of a client ID, a client thread ID, and a continuously numbered request ID. It is needed for two purposes. First, it is required for *request duplication suppression*. As described in Section 3.1.3, failures can trigger the re-invocation of a method. A unique request identifier allows the servant group to detect such duplicated invocations. Second, unique request identifiers are also needed for *logical thread identification*. This requirement is not only needed for a single-logical-thread scheduling strategy (SLT, see Section 3.2), which requires information about the logical thread a request belongs to. In a leader-follower synchronisation model, such as Basile's loose synchronisation algorithm (LSA), request-handling threads similarly need an ID that is consistent in all replicas.

Reentrant Locks

The thread identification that is provided by the context handler is also needed for supporting reentrant mutex locks. A mutex lock is reentrant if it can be acquired multiple times by the same thread. A thread can invoke nested invocations, which, ultimately, can lead to a circular invocation chain that invokes a method at the originating object. This invocation logically belongs to the same thread, even if it is handled locally by a new low-level thread. The logical thread identification provided by the context handler can indicate that two low-level threads belong to the same logical threads. Thus, reentrant lock request can be granted to low-level threads using this logical thread identification.

Marshalling

The *Marshalling* component is responsible for the serialisation of invocation data and the deserialisation of replies. The marshalling is identical to that of a standard CORBA stub. Corresponding components exist at the client and the server side.

Communication

The *Communication* components in all fragments provide the interaction between fragments. The component depends on the concrete replication and consistency model. This thesis uses a totally ordered group communication system for strictly consistent active replication. The *Communication* component represents an abstraction that makes the fragment implementation independent from a specific group communication system. Section 4.2 discusses the influence of semantic properties of the group communication on the tasks of the *Context Handler*. The implementation supports both *open* and *closed* group communication systems. A closed system allows only group members to send group messages, whereas an open system also permits external senders of group messages.

Closed Groups

If a closed group communication system is used (e.g., JGroups [Ban98]), only replicas use group communication to exchange messages among themselves. Access fragments need to communicate with a *gateway* to access the replica group; each *Communication* component in a replica fragment can act as such a gateway, passing requests to the local group communication. With the gateway approach, the *Communication* element in the access replica is responsible for transmitting calls to a single available replica. If this replica fails, the *Communication* component transparently reconnects to another replica fragment and reissues the call. If the call has already been processed by the replicas, the duplication is detected and the invocation result is returned from a cache.

Finally, the *Communication* component in replica fragments places all incoming messages into a totally ordered queue for subsequent processing by the upper layers.

Open Groups

The alternative is to use an open-group system, such as the Aspectix group communication system (AGC), which is described in Chapter 6. This latter approach can improve performance (e.g., the access fragment may directly multicast its request to all replicas), and is also beneficial for the support for Byzantine fault tolerance, as access fragments may receive replies from multiple replicas and can use voting.

4.1.3 Benefits

Evaluation

Fragmented objects can be used to integrate replication support into distributed object middleware. The FT*flex* architecture uses this approach to provide fault tolerance. The PECSAR criteria, as defined in Section 3.1.1 of this thesis, are used in the following to evaluate general aspects of this architecture.

Portability

The fragmented-object model allows supporting fault tolerance without modifying the middleware system. The FT*flex* replication system is portable across platforms that support such a model; it requires only that the middleware be able to understand remote references that address fragmented objects by dynamically loading the code specified by the reference. This approach is more flexible than, e.g., a standard CORBA system, in which a deviation of the standard stub-servant structure is more difficult to realise. The support for a fragmented-object model is easily added to existing middleware systems. The Aspectix middleware, which is used as the basic infrastructure for FT*flex*, demonstrates such integration into the JacORB CORBA system. Two additional prototypes show that this concept can also be used to add fragmented objects to Java RMI [KKSH05] and to .NET Remoting [RDH05].

Efficiency

In the fragmented-object model, the fragment code is directly loaded at the client side. The client application directly invokes methods at the access fragment, which handles the interaction with the replication group. The direct interaction with the local fragment avoids any indirection or interception and results in optimal efficiency. Another advantage is that the fragment code can be optimised on a per-object basis. It is even possible to have multiple variants for the same object type. This differs from traditional approaches, which use static, fixed code for replication. The use of semantic annotations and the automated generation of fragment code are discussed in Section 4.4.

Client Transparency

The FT*flex* architecture provides full client transparency on the basis of the CORBA programming model. The interface of a traditional stub and that of a local access fragment are identical, and the access fragment is automatically loaded by the Aspectix middleware.

Servant Transparency

The fragment architecture of FT*flex* does not provide servant transparency. It faces the same limitations that characterise other fault-tolerance platforms, such as the requirement of explicit state-transfer support and the restriction to deterministic behaviour. As a partial compensation, the next sections will describe a few mechanisms that simplify the implementation of replicated objects and the re-use of existing object implementations. The most important provisions are a partial automation of state transfer and deterministic multithreading.

Adaptivity and Reconfigurability

The use of fragmented objects simplifies the integration of runtime adaptivity and reconfigurability. The current prototype considers adaptivity only for low-level consistency mechanisms. These mechanisms are provided at the group-communication level (see Chapter 6). For client-triggered reconfigurations, the fragmented-object model of Aspectix provides a *policy interface* that allows the specification of configuration requests at runtime. The FT*flex* replication infrastructure uses a simplified version of this policy interface, as described in Section 4.5.2.

4.2 Consistency Challenges

Overview

This section is about consistency issues that relate to the replica state, the validity of external interactions, and the return values sent back to clients. It discusses the necessity of total messages ordering at the group-communication level and analyses the challenges that are caused by node failures. For various failure models and group communication properties, the FT*flex* architecture provides appropriate mechanisms that ensure replica consistency as well as consistent external interactions and client return values.

4.2.1 Message Ordering Semantics

Total Order

This thesis assumes that totally ordered group communication (i.e., abcast) is used in order to provide active replication of objects. Weaker ordering semantics, such as causal (cbcast) and FIFO (fbcast) message order, are not sufficient in most cases. If two clients independently invoke replica methods that both modify a state variable i (for example: (a) $i := i + 1$ and (b) $i := i * 2$), the result is deterministic only if the order of method invocations is the same on all replicas. Only abcast provides such a total order for independent requests from multiple clients.

Weaker Ordering Semantics

The relative order of some requests is irrelevant. This is the case for requests that access disjoint parts of the object or that have the same effect on the object state regardless of their execution order (such as two increment operations on a state variable). At the group communication level, however, such semantic properties of requests are usually unknown. Because of this lack of knowledge, there is no simple alternative to using totally ordered group communication for all requests.

Concurrent Execution

Totally ordered group communication is necessary not only for a strictly sequential execution of object methods. In Chapter 5, this thesis presents strategies for multithreaded execution of client requests. Such a multithreaded request handling can result in the execution of multiple requests without adhering to a strict global order. The scheduling strategies enable the execution of multiple threads in a way that guarantees consistency even in spite of such a concurrency. Despite of the concurrent execution, the scheduling algorithms all require a global order of incoming requests, and thus do not work with weaker ordering semantics at the group-communication level.

4.2.2 Consistency Challenges with Failures

Overview

The FT*flex* architecture is based on the active replication of objects. This replication style assumes deterministic object behaviour and thus achieves consistent object state by redundant execution of the same methods in all non-faulty replicas. The issue of consistency, however, not only addresses the internal state of non-faulty replicas. It also concerns external effects of faulty replicas. While the internal state of a replica after a failure can be considered irrelevant, what matters in practice are potential interactions of such nodes with the environment in form of nested invocations to other nodes and replies sent to clients. The effects of these interactions still remain visible after a failure. Both non-benign and benign failures require adequate provisions in the replication infrastructures:

Byzantine Failures

External Inconsistencies Caused by Byzantine Failures

The necessity of provisions for faulty nodes is more obvious for non-benign failures. In a Byzantine failure model, a faulty node can exhibit arbitrary behaviour, which poses two challenges. First, a faulty node can send arbitrary reply messages to clients that have sent a request. The client obtains a correct reply only if the replication infrastructure makes sure that the reply received by the client originates from a non-faulty node. Second, the faulty node can issue invalid nested invocations. Such invocations can have unwanted external side effects that seem to originate from a client request.

Benign Failures

Uniform and Non-Uniform Multicast with Benign Failures

The potential problems that arise from benign failures depend on the message delivery semantics. Group communication can be implemented with uniform and non-uniform message delivery semantics, as defined in Section 3.3.1. In non-uniform multicast, if a set of nodes delivers a message m and all nodes of this set subsequently fail, there is no guarantee that the surviving group nodes will deliver m. This guarantee is strictly weaker than that of uniform multicast. Uniformity requires that if *any* node delivers a message—even if it subsequently fails—all other non-faulty nodes will deliver this message. This means that the delivery of a message must be delayed as long as it is not sure that all other non-faulty group nodes will receive the message. As a result, it typically has a higher latency than non-uniform group communication.

Practical Impact

The difference between both variants is only minor at a first glance, but in practice has an important impact on object replication. Non-uniform communication is sufficient if the delivery of a message m affects only the internal state of the receiving node. The difference, however, matters if the message reception causes external side effects. Such side effects can be either subsequent external interactions triggered by the message reception or a reply message sent back to the sender of m. In the non-uniform case, a node P_i can, for example, pass a reply to the sender of m. If P_i subsequently crashes, it is possible that the surviving nodes do not know about this reply; they may even produce a different reply, resulting in an inconsistent client state.

Using Uniform Group Communication

A benign failure model permits the use of a uniform group communication system. This approach has the advantage that no consistency violations due to node failures can occur. An operation returns a value to the client and issues a nested invocation only if it is guaranteed that the replica group will execute the operation, even in case that some replicas fail. The Aspectix group communication system (AGC), which is discussed in Chapter 6, is able to provide such a uniform semantics for benign crash-stop and crash-recovery failures. A uniform group communication system thus avoids all problematic consistency problems.

Using Non-Uniform Group Communication

Using uniform group communication is not always a possible choice. It might be desirable to use an existing group communication that lacks support for uniform group communication (this is, for example, currently the case for the popular JGroups system [Ban98]). A further disadvantage of uniform group communication is its performance. A message is delivered only if it is certain that all other nodes will deliver that message after a crash. This requires at least one additional message delay before a message may be processed, as sufficiently many nodes have to confirm that they know about the message delivery. Hence, the use of non-uniform group communication is desirable in practice for efficiency reasons. If a non-uniform group communication system is used, consistency problems can arise because of external interactions of a node that subsequently fails, and thus the replication infrastructure has to provide adequate means for avoiding these problems.

4.2.3 FT*flex* Solutions for the Consistency Challenges

Overview

The FT*flex* replication infrastructure provides flexible support for various failure models and various group communication systems. The Aspectix group communication system (AGC), which is presented in Chapter 6, offers uniform or non-uniform semantics, depending on the failure model and on the selection of the internal algorithms. In addition, FT*flex* can also be used with other group communication systems such as JGroups [Ban98]. FT*flex* thus needs to provide a solution that is able to support benign failures with uniform and non-uniform group communication semantics as well as Byzantine failures. The use of uniform group communication in a crash-stop failure model is efficiently enabled by disabling any additional measures. For all other cases, FT*flex* provides the subsequently discussed solutions.

Solutions for a Benign Failure Model

Without External Side Effects: No Semantic Problem

If a request m, received by node P with non-uniform semantics, causes only internal state modifications at node P, no consistency problems can arise. The state modification is a local action at node P that is not visible on other nodes. If node P fails, its local state becomes irrelevant for the remaining system. In this situation, non-uniformity does not cause any semantic problems. In practice, however, most requests will have some external side effects. The limitation to internal state modifications is violated if (a) the execution of the requests causes a nested invocation or (b) the request returns some value to the client.

Nested Invocations

If the execution of a request m at replica P causes a nested invocation, and P subsequently fails, it may happen that the remaining replicas execute request m in a different relative order to other requests. This can, for example, have the effect that the remaining replicas invoke the nested invocation with different arguments, resulting in an inconsistency. A practical solution, as implemented by FT*flex*, is to delay the nested invocation as long as it is not yet known that all replicas will process the requests in the same order and thus issue the same nested invocation.

Inconsistent Replies after Failure

If the client receives a reply from the replica group, this reply can originate from a replica that later fails. The non-faulty replicas could later produce a different reply (for example, because they process the requests in a different order). In this case, the client state becomes inconsistent with the replica group. The problem is avoided if the infrastructure delivers the reply to the client only after it is sure that all nodes will process the same request.

Solutions for a Byzantine Failure Model

Challenge

Byzantine nodes create a similar, albeit more severe problem, as faulty nodes may show arbitrary erroneous behaviour. Again, it is necessary to ensure consistent nested invocations and client-side consistency. In the following, it is assumed that the number of node failures is limited by some value M.

Nested Invocations

Supporting a Byzantine failure model for consistent nested invocations requires support in the target of the invocation. As a faulty node can have arbitrary malicious behaviour, the target must verify that the invocation indeed originates from a non-faulty node. Assuming a maximum of M faulty nodes in a replica group with N members, receiving the invocation requests from $M+1$ group member ensures that at least one non-faulty node supports the request. The basic design of FT*flex* thus is to send an invocation request from all source replica nodes and verify the reception of at least $M+1$ identical requests at the target of the invocation.

Client Return Values

The same principle is also used for producing consistent return values for clients. If at least $M+1$ replicas have generated identical replies, it is certain that at least one such reply originates from a non-faulty replica. Consequently, all other non-faulty replicas will do the same because of the guarantees of non-uniform semantics.

Replica-Side Verification

The verification of having $M+1$ identical invocation requests or reply values can also be done within the replica group. For this purpose, the replica group can internally collect $M+1$ identical messages from replica nodes, with an individual signature from each replica. In this case, only a single message needs to be passed to the invocation target or client, carrying $M+1$ signatures as proof that at least one non-faulty node supports that message. This approach simplifies the external communication of the replica group. It, however, adds the overhead of using digital signatures, which not only causes additional processing time, but also requires a public-key infrastructure that enable clients and invocation targets to verify signatures of all replicas.

Increasing Efficiency with Messages Digests

Sending multiple invocation or reply messages can cause a performance bottleneck; this is especially the case for large reply messages. The recipient of the message only needs a single message; all further messages are only used to verify that other replicas have created the same message. It is thus sufficient that only a single replica sends the full message, while all other replicas send only a short message digest to proof the creation of the same invocation or reply message.

4.2.4 Read-Only Operations

Common Approach: Identical Handling of all Methods

There are various options for how to execute read-only operations, i.e., invocations of object methods that do not change the object state and do not issue nested invocations that have external side effects. Most existing replication infrastructures do not distinguish between read-only and modifying operations.

For an active replication style, this means that read-only requests are distributed to all replicas by totally ordered group communication.

Executing Read-Only Methods at Just One Replica

One apparently obvious optimisation is the execution of read-only methods at only a single replica. This approach requires the availability of semantic information about a method being read-only; in FT*flex*, such information can be provided by semantic annotations, as described in Section 4.4. Such handling of read-only operations not only eliminates the cost of the reliable multicast operation, but also saves computational resources by executing the method only on a single node.

Semantic Problem

The drawback of the simple solution above is that the semantic properties of the invocation order are lost. If a “shortcut” is used for read-only operations, such an operation might be executed earlier than a modifying operation that was issued before. This order violation can even disable simple read-your-writes semantics. A modification operation can deliver a result to the client as soon as it is sure that all replicas will *eventually* execute this request. A later read-only operation of the same client can be executed on a replica that did not yet execute the modification. Hence, the read-only operation will read an outdated state.

Solution 1: Update All Strategy

The first solution to this preceding semantic problem is the use of an update-all strategy. This means that the invocation of a modifying method only returns to the client after updating *all* replicas. This ensures that a subsequent read operation will always read the most up-to-date state. The disadvantage of this approach is the dependency on the availability of all nodes.

Solution 2: Check Invocation Counter

The FT*flex* architecture favours a second solution, which uses the invocation context to detect read access to outdated states. The context information, which is added by the *ContextHandler* component, consists of a unique client ID and a unique invocation ID. All invocations of methods at a specific remote object are numbered sequentially. A replica that receives a read-only operation can use the invocation counter to check whether it already has received all prior invocations from the same client. As a result, a read-your-writes semantics is maintained.

Limitations of Solution 2

The limitation of the second solution is that it ensures read-your-writes semantics only between two directly interacting objects, but not for arbitrary transitive interactions. For example, an object A might invoke a modifying operation at object B, which in turn invokes a modifying operation at object C. The invocation of A can return as soon as B and C will eventually execute it. This means that there can be replicas of C that have not yet received the modifying operations; these replicas don’t known anything about the pending modifying operation. If A later invokes a method at object C, the infrastructure cannot guarantee that this read-only operation is performed after the preceding modification.

Solution 3: Propagate Explicit Version Numbers

An improved solution uses the propagation of explicit version numbers. This strategy is not yet implemented by the FT*flex* prototype. The basic idea of this variant is that each replica group knows its “version number”, denoted by the local sequence number of the last successful modifying operations. This version number is propagated to the client with replies, and back to replicas at subsequent invocations from the client. In case of nested invocation, the version numbers of all accessed replica groups are collected. The approach is similar to the use of logical vector clocks; these clocks enable the exact detection of causal relations. If a read operation is scheduled at a host and this host detects that the client knows about a more recent operation, the host suspends the read operation until it has executed sufficient modifications and reaches the same local version number. The disadvantage of this solution is the amount of data that needs to be transferred with all requests and replies.

Solution 4: Don't Care About Read-Only Property

The last option for handling read-only operations is to treat them as modifying operations. This variant eliminates all semantic problems, but also fails to draw any benefit from the semantic knowledge.

4.3 State Transfer

Basic Purpose

This section describes the realisation of state transfer in the FT*flex* architecture. A state transfer mechanism retrieves the current state of a replicated object from existing replicas and initialises a new replica with this state. Initialising a new replica at runtime is necessary for increasing the total number of replicas and for recovering crashed replicas.

4.3.1 Infrastructure Support in FT*flex*

Manual State Serialisation

Most replication middleware systems (e.g., FT-CORBA) use a manual approach for state serialisation and deserialisation. This means that the developer has to add methods for obtaining and for setting the replica state for each object implementation. The advantage of this approach is that the developer gets full control on how to serialise the object state. For this reasons, FT*flex* also supports this simple, lightweight, and flexible approach. In Section 4.3.2, we will briefly discuss extended variants.

Challenges of the Infrastructure Integration

The integration of support for state transfer in a replication infrastructure faces several challenges:

- The state transfer must be synchronised with the processing of client requests.
- The state transfer must be fault tolerant; this means that it should be able to cope with the failure of a replica during the state transfer.
- The state transfer should have minimal impact on the normal operation of the replicas (i.e., on the processing of client requests).

In the following, we first assume a simple model with a strictly sequential execution of requests. Later, this model is extended to multithreaded request execution. Figure 4.4 illustrates the state-transfer process for a joining replica. First, the new replica joins the replica group at the level of the group-communication system. The successful integration of the new node in the communication group is signalled by a *new view* event in the sequence of incoming messages.

Synchronised Transfer

The reception of the *new view* event defines the point in time on which the state of the existing replicas needs to be captured. The new replica will receive all client requests after this event, and thus needs the state that the replicas have before the next request. A trivial strategy for a coordinated state transfer would be to block the processing of new requests until the state is successfully transferred to the new replica. Such an approach, however, has its deficiencies. Blocking all replicas for the entire duration of the state transfer has a serious impact on performance. Furthermore, if the new replica fails during the state transfer, additional measures are necessary to avoid an infinite deadlock (the

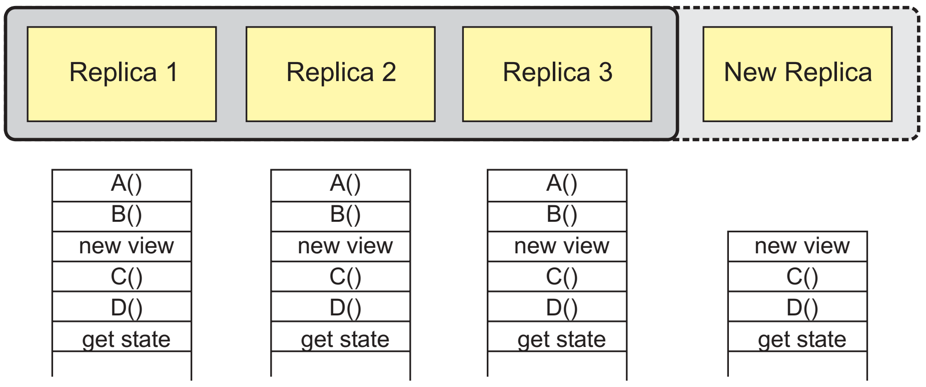


Figure 4.4: State transfer for joining replicas

state transfer will never complete because of the failure, and the replicas are blocked until the transfer completes). For this reason, the FT*flex* architecture uses a slightly different strategy. Upon reception of the *new view* event, the `get_state()` method of the replica implementation is called, triggering the serialisation of the state into a temporary variable. This variable can later be transferred to the new node, while all replicas can resume processing requests as soon as the state serialisation has finished. This minimises the impact of state transfer on the execution of other requests.

Fault-Tolerant Transfer

The state transfer must also be able to cope with node failures. On the one hand, existing replicas can fail during the state transfer. These failures should be tolerated on the basis of the same failure semantics as the group uses to handle ordinary client requests. This means that with a crash-stop or crash-recovery failure model, the state of a single non-faulty replica needs to be transferred to the new node; with a Byzantine failure model, multiple nodes must participate in the state transfer. The straightforward approach of FT*flex* is to handle state-transfer requests in the same way as remote method invocation requests. Therefore, a new node that joins a replica group invokes a remote state-transfer method at the replica group after receiving the *new view* event. This remote invocation transfers the replica state from the temporary variable, which has previously been created in all replicas at the reception of the *new view* event. As the state-transfer method does not access the replica implementation (but only the temporary variable), this request can be executed "out of order" concurrently with the execution of other client requests.

4.3.2 Extensions

Overview

The basic state-transfer support in FT*flex* can be extended for supporting parallel state transfer from multiple nodes, it can be partially automated, and it can be combined with multithreaded execution of client requests.

Extensions for Distributed State Transfer

The approach described above first serialises the state to a temporary variable, and then transfers the replica state via a remote invocation at the replica group. Zeman [Zem06] describes variants that enable a direct transfer without creating a temporary copy of the state and support parallel state transfer from multiple replicas on the basis of peer-to-peer technology. The details of these extensions are beyond the scope of this thesis. The main benefits of the variants are improved efficiency for the transfer of huge object states (hundreds of megabytes to gigabytes range) and faster state transfer over communication channels with small capacity.

Extensions on the basis of IDL Value Types

The manual implementation of `get_state()` and `set_state()` methods complicates state transfer between independent implementations (the developers have to agree on a common state exchange format), it causes additional work, and thus adds additional complexity to the object development. Previous work in the Aspectix project supports object migration with automated support for state transfer via value types [KSH05]. This mechanism can also be applied to state transfer for replicated objects. A replicated service is defined as an IDL value type that defines the object state in terms of IDL data types. Such a value type can be automatically serialised, which provides a mechanism for state transfer automation. Using this approach requires no modification to the FT*flex* architecture, as long as the required `get_state()` and `set_state()` methods are provided.

Benefit: Improved Support for Heterogeneous Platforms

The advantage of the approach on the basis of IDL value types is that the generic state definition in IDL allows an easy exchange of state between heterogeneous replica implementations. Several implementations of replicas, e.g., in different programming languages, with transparent state transfer between them are feasible this way. Such an implementation redundancy is useful in a Byzantine failure model, if multiple independent replica implementations are provided to tolerate even implementation faults.

Multithreaded Execution of Requests

In Section 5.3, this thesis describes the support for multithreaded request execution in the FT*flex* architecture. With such use of multithreading, it is not sufficient to coordinate the state transfer with the preceding and following client request. Instead, it is necessary that, at the moment of the state transfer, no more threads of previous requests be active in the replicas. For this purpose, the multithreading support must provide the functionality of waiting for the termination of all active threads. This support is discussed in more detail in Chapter 5.

4.4 Semantic Annotations

Augmenting IDL Definitions with Semantic Annotations

The code generation process of FT*flex*, which is described in detail in Section 4.1.1, can automatically generate access fragments and replica fragments on the basis of IDL interface definitions, similar to the creation of stubs and skeletons by a traditional CORBA IDL compiler. The flexibility of the code generation process is increased if the developer gets the possibility to influence the generated code. For this reason, FT*flex* integrates *semantic annotations* into the code generation process and thus offers a means to improve and customise the replication mechanisms. Semantic annotations can specify behavioural and structural properties of object methods. Behavioural annotations express the developer's knowledge of properties that allow an optimisation of remote invocations. Structural annotations specify the distribution of functionality between client-side access fragments and replica fragments.

```
• readonly
• parallelizable(methodlist)
• local
• internal
• intercepted
```

Figure 4.5: Annotations supported by FT*flex*

4.4.1 Generic Specification of Annotations

Generic Support

The support for annotations in the FT*flex* architecture is generic. This means that the integration of new annotations is easily possible. Technically, all developer annotations are parsed by the code generation tool and then evaluated according to the mapping specification. The mapping specification can be extended to create custom code for new annotations. The current prototype uses `#pragma` statements to embed annotations directly into IDL specification.

Alternative to `#pragma` Statements

One might doubt that the inclusion of semantic annotations into the IDL specification is a good idea. In current object-oriented middleware systems, the IDL has two distinct purposes: (a) it defines a contract for client developers, specifying the interfaces that can be used for interactions, and (b) it is used as the basis for the generation of stub and skeleton code. In traditional systems such as CORBA, the same IDL information is used for both purposes. The semantic annotations, however, are required only to influence the code generation process, but should not affect the client-side interface. A separation of annotations and interface specifications could thus be justified. The current FT*flex* prototype, however, uses the `#pragma` approach for the specification of semantic knowledge, as it was not difficult to add the evaluation of `#pragma` statements to the existing configurable IDL*flex* code generation tool. Furthermore, an IDL specification of a service without annotations, which is sufficient for client application developers, can easily be generated from an annotated IDL.

4.4.2 Supported Annotations

Overview

The current prototype defines a mapping for the set of annotations shown in Figure 4.5. This set includes behaviour annotations (`readonly` and `parallelizable`) as well as structural annotations (`local`, `intercepted`, and `internal`).

The fragment code generator was extended to support semantic annotations in IDL files, expressed as `#pragma annotate` statements. Within a custom mapping specification, these annotations are evaluated and used to control the code generation process.

`readonly`

A method marked as `readonly` does not modify the relevant replica state. Within the limitations discussed in Section 4.2.4, it is possible to invoke a read-only method on a single available replica, instead of executing it in total order on all replicas.

If at least one `readonly` method is present, code is generated for the *Communication* component that examines all invocation requests. If read-only optimisation is enabled and a method marked as read-only is requested, a replica fragment will pass the request directly to the upper layer, bypassing the emission via totally ordered group communication.

parallelizable

A method marked as `parallelizable` with respect to a set of other methods can be executed in parallel with the specified list of other methods. With this annotation, the developer guarantees that any execution of a set of parallelisable methods with multiple threads will have the same effect, regardless of the thread scheduling and the relative execution speed of the threads. For example, methods that *access only disjoint parts* of the replica state are parallelisable. Furthermore, methods that access a common part of the state, protected by a mutex, are parallelisable, if the order in which the mutex is granted to threads does not influence the result. This is the case for *idempotent* operations.

The *Scheduling* component in replica fragments is informed about all methods that are marked as `readonly` or `parallelizable`. The component uses an instance of a deterministic thread-scheduling algorithm, as described in Chapter 5. The algorithm instance can use the annotation information to optimise the scheduling of such methods.

local

The implementation of a method marked as `local` is implemented in the client-side fragment. As a result, methods that need no access to the replica state can be executed locally at the client, while still being conceptionally part of the distributed object. Such a method implementation cannot access the replica state directly; however, it may invoke internal methods at the replica fragments.

internal

A method marked as `internal` is not part of the client-side object interface. Such methods are implemented at the replica side and accessible via remote invocations. They are not intended for direct client invocations, but they can be invoked by the local methods of the client-side access fragment.

intercepted

A method marked as `intercepted` will execute custom code at the client-side before and after invoking the remote method at the replica group. The interceptor code can be used for local preprocessing, for caching, or for the accumulation of multiple client invocations into a single remote invocation to the replica group.

The current prototype requires that developers manually implement the additional client-side code. For each method annotated as *intercepted*, an abstract method is created in the access fragment; the developer has to provide the actual method implementation.

Intercepted Method vs. Pair of Local/Internal Methods

Intercepted methods are provided mainly for convenience reasons. In principle, it is possible to simulate an intercepted method with a local and an internal method. In such scenario, the local method would provide the client-side interception functionality and then invoke the remote internal method. An intercepted method differs from a pair of local/internal methods in terms of method naming in the fragment implementations. An intercepted method uses its IDL name for the replica-side implementation; the replica implementation is thus identical to a non-replicated object. With the alternative approach, the local method and the internal method must have different names. As the local method implements the IDL function that the client uses, the replica-side internal method must have a different name. In other words, the intercepted construct maintains servant transparency, while a local/internal pair does not. On the other hand, local and internal methods provide a higher degree of freedom.

```
module Library {
#pragma annotate parallelizable(borrowBook), readonly
   User lookupUser(in String name);

#pragma annotate parallelizable(lookupUser)
   void borrowBook(in User user, in Book book);

   // ...
}
```

Figure 4.6: IDL definition of the library example with annotations

4.4.3 Use Cases

Example 1: Library Object

The following examples demonstrate typical use cases for semantic annotations. The first example uses an object-oriented implementation of a library application. Figure 4.6 shows a part of the IDL interface of a library object. The IDL defines two methods; the first one permits the client application to look up a library user by name, and the second one offers the possibility to borrow a book.

Annotations `parallelizable` and `readonly`

The example demonstrates the use of the `parallelizable` and `readonly` annotations. The lookup of a user does not modify the internal state of the library object, and thus the lookup can be executed by just a single replica, instead of requesting the method execution at all replicas. Under the assumption that the information about which book is borrowed by which user is not stored in the user information (but instead in the book database), the `borrowBook()` method does not modify any state information that is accessed by `lookupUser()`. It is thus possible to execute both methods concurrently without risking conflicting state modifications. The developer can provide this information to the FT*flex* infrastructure with `parallelizable` annotations at both methods. The `borrowBook()` method cannot be annotated as `readonly`, as it modifies the internal object state.

Example 2: Credit-card Processing Service

The second example uses a credit-card processing service, which clients can use to handle credit-card transactions. The simplified IDL in Figure 4.7 offers methods for charging money on a credit card and for verifying the validity of a credit card; `validate_card_checksum()` only verifies whether the checksum of the card number is valid (which can be confirmed by a simple numerical calculation), while `validate_card()`, on the other hand, interacts with the card issuer to verify whether the card indeed is valid (e.g., has not been reported stolen).

Annotations `local` and `intercepted`

Structural annotations can be used to implement an improved version of the credit-card processing service without changing the client interface. First, the `validate_card_checksum` method only performs a calculation without accessing any service state. It thus can be implemented directly in the client-side fragment without having to use a remote invocation. For this purpose, the `local` annotation can be used. Second, the `validate_card` method can be marked as intercepted. Interception can be used to call `validate_card_checksum` first before issuing the remote invocation that checks the validity with the card issuer. If the checksum is false, the card is invalid, and validation can be aborted with the `CardNotValid` exception without remote interaction.

```
interface CC_processor {
  transaction_id charge(in carddata card, in float amount)
      raises (CardNotValid, TransactionFailed);

#pragma annotate local
  boolean validate_card_checksum(in card_data card)
      raises (CardNotValid);

#pragma annotate intercepted
  boolean valiade_card(in card_data card)
      raises (CardNotValid);

};
```

Figure 4.7: IDL definition of the credit-card processor example with semantic annotations

```
interface GenericFragmentFactory {
  Object create_initial_replica (
         in Key k,
         in Credentials cred,
         in Criteria the_criteria )
     raises (NoFactory, InvalidCriteria, CannotMeetCriteria);

  void create_replica (in APXObject fo,in Criteria the_criteria )
     raises (NoFactory, InvalidCriteria, CannotMeetCriteria);
};
```

Figure 4.8: Interface of the generic fragment factory

4.5 Management of Replication Groups

Overview

Similar to other fault-tolerant middleware infrastructures, Aspectix uses the factory pattern to create and set up replicas. Replication groups are implemented as self-managing entities; this design reduces the complexity of the necessary infrastructure compared to other systems that require a dedicated replication manager. In addition, the management automatically benefits from the same fault-tolerance mechanisms as the replicated object itself.

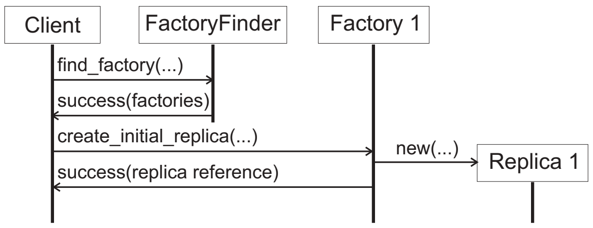
4.5.1 Runtime Infrastructure

Finding the Initial Factory

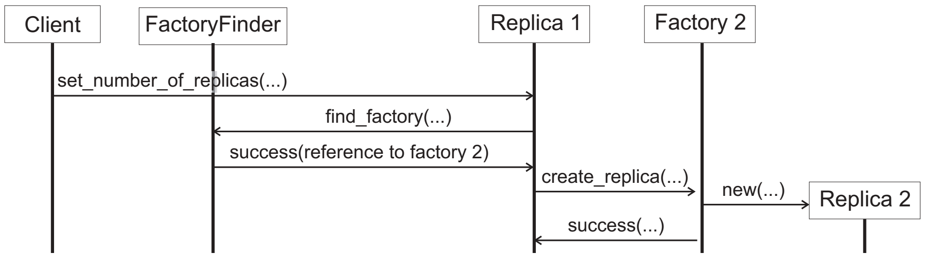
Starting a new replicated service involves several steps. First, a factory has to be acquired via a factory finder. A factory finder represents a search scope for possible places of execution, corresponding to the definition by the CORBA Life Cycle Specification [OMG02]. Currently, the factory finder is implemented in a straightforward way in plain CORBA and well-known on every node within a domain. Thus, a node can register its local factories and can look up factories from all other nodes. Multiple factory finders can be provided for fault tolerance.

Replica Creation by the Factory

The generic factory for fragment creation offers two methods (Figure 4.8): one for setting up an initial replica of a replicated fragmented object and another



(a) Creation of first replica



(b) Creation of additional replicas

Figure 4.9: Creation of the first replica and of additional replicas

one for setting up additional replicas. After a lookup of one or more factories via the factory finder, one of them is requested to instantiate the first replica via `create_initial_replica()` (see Figure 4.9a). The factory creates the initial replica and activates the fragment object. Afterwards, the object is returned to the calling client; implicit binding causes the instantiation of a local access fragment at the client side. The result is a simple client-server structure with only one replica. The management code within this replica is able to control the creation of additional replicas.

Management Interface

A management interface of the fragmented object is used to increase the desired number of replicas (see Figure 4.9b). An increase triggers the existing replica group to add the necessary number of additional replicas. The replica-side fragment contacts the factory finder to request additional factories. In the next step, a reference to the fragmented object is passed to a factory. At the factory side, the fragmented object is transparently bound by the middleware, which loads the initial fragment. Under control of the factory, the local fragment is reconfigured to be a replica fragment. The state of the existing replica group is transferred to the new replica as described in Section 4.3. The addition of replicas is repeated until the desired replication level is reached or until no additional factories are found.

Handling of Failures

The failure of a replica in the group is detected by a failure-detection mechanism at the group-communication level. After detecting a failure, the replica group automatically sets up a new replica in the same way, as long as another factory is available.

4.5.2 Dynamic Runtime Reconfigurations

Aspectix Policy Configurations

The core of the Aspectix middleware provides complex facilities for dynamically

```
interface Reconfigurable {
    void set_policy(in String key, in String value);
    String get_policy(in String key);
    sequence<String> list_policies();
}
```

Figure 4.10: `Reconfigurable` interface

configuring fragmented objects on the basis of policies[1]. Aspectix provides powerful mechanisms for handling policy reconfigurations. Fragment implementations can receive notification about policy changes requested by the client, and can use them to trigger reconfigurations. Furthermore, fragments can verify whether a desired policy configuration is currently possible. If not, a fragment can inform the application about this problem via registered callback handlers.

*FT*flex *Policy Configurations*

The FT*flex* infrastructure uses only a simplified version of the policy mechanisms of Aspectix. FT*flex* uses only string values as policy values instead of arbitrary complex data types. It also does not make use of policy verification and callback notifications back to the client. Instead, policy updates by the client are simply passed to the fragment implementation. This simplification is motivated by the fact that the extended Aspectix features are not essential for the replication support. Furthermore, the simplified policy configuration model makes it easier to port the FT*flex* infrastructure to other middleware systems based on fragmented objects, such as our FORMI infrastructure [KKSH05].

`Reconfigurable` Interface

Figure 4.10 shows the IDL specification of the `Reconfigurable` interface. If an object developer wants to enable client-side reconfigurability for some object, this object has to inherit the `Reconfigurable` interface at the IDL level. The code that is generated for the access fragment contains generic functionality to forward policy updates to the infrastructure and to the fragment implementation.

4.6 Evaluation

Overview

Replication on the basis of fragmented objects provides flexibility. This flexibility manifests itself in various ways. First of all, an infrastructure for fragmented objects can support replication without internal modifications to the middleware. Currently, many middleware systems support replication only after intrusive extensions. Furthermore, the FT*flex* architecture provides a code-generation tool that automatically creates fragment code. The tool offers the flexibility to structure applications in a way not possible with traditional remote objects. This flexibility can be used to provide parts of the replica code at the client-side, while maintaining object identity. Section 4.6.1 discusses this advantage using the example of a source-code version control system. In addition, the developer can influence code generation with semantic annotations. Section 4.6.2 shows

[1]In the original Aspectix middleware, these policies are called *aspects*. With the rise of *aspect-oriented programming*, the term *aspect* today is usually understood to mean a portion of program code that is scattered around multiple functional modules of an application (i.e., "cross-cutting"), but belonging to a single (functional or non-functional) concern. An Aspectix aspect, on the other hand, is a generic piece of data that influences the internal functionality of a fragmented object.

measurements that demonstrate the increase in performance that the developer can obtain with these annotations.

4.6.1 Client-Side Code

Client-Side Code in Source-Code Version Control System

The development process of a replicated service on the basis of fragmented objects allows the implementation of client-side code. Using the FT*flex* infrastructure, Baumann [Bau06] has implemented a replicated source-code version control system. This system internally uses the same mechanisms as *git*, the source-code version control system that is currently used by the Linux kernel[2].

Functionality of a Version Control System

A version control system stores the revision history of all file versions of a project in a central *repository*. The most important functions of such a system are mechanisms to *check out* a copy of the repository files into a client-side copy, to *commit* local modifications to a new revision in the repository, and to *update* the local copy to the most recent version in the repository. The implementation of a version control system consists of *server-side functionality* for managing the repository and *client-side functionality* for interacting with the (usually remote) repository and manipulating the local copy.

Realisation of Client-Side Code in FTflex

With FT*flex*, a source-code repository of the version control system can be represented by a fragmented object, identified by a CORBA IOR. Fragmented objects allow the developer to provide client-side code for an object implementation. This code can implement functionality for reading and updating the client-side copy. This means that functionality for modifying local files can be implemented as conceptional part of the fragmented object. Current middleware systems that are based on the notion of *remote* objects cannot provide such a feature.

Related Developments: AJAX

Related to this, it is an important observation that remote services that use client-side computations are getting more and more important in other domains. Recently, AJAX (Asynchronous JavaScript and XML [Gar05]) has gained much popularity. AJAX enables the provision of client-side functionality as part of interactive web applications and thus can be used to increase the usability and interactivity of such applications. The FT*flex* approach allows a similar application structure for CORBA-based distributed objects.

4.6.2 Benefits from Read-Only Annotations

Annotation of Read-Only Methods

Many distributed objects provide remote methods that do not modify object state. For example, the source-code version control system discussed later offers several methods that query the current state of the remote repository and retrieve updates. As these operations only read the server-side repository, they can be executed by a single replica. Traditional object middleware is unable to distinguish between read-only and modifying requests due to the lack of semantic knowledge. FT*flex* introduces semantic annotations to express such knowledge, which developers can use for optimisation.

Experimental Evaluation

The following measurement demonstrates the difference in invocation time between a read-only method and a modifying method. A single client accesses a replica group with the number of replicas increasing from one to five nodes. The measurement has been made on a set of PCs with a AMD Opteron 2.2 GHz CPU,

[2]see `http://git.or.cz/` (valid 2006-09-16)

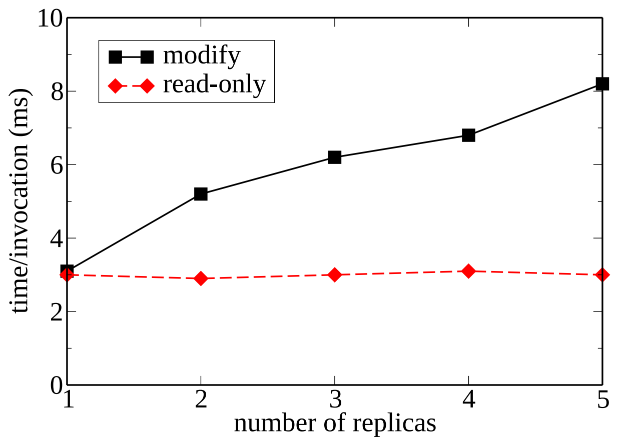


Figure 4.11: Efficiency gain with read-only annotations

using Linux kernel 2.6.17 and connected by a 100 MBit/s switched Ethernet network. The current prototype of the Aspectix middleware was used on the basis of Sun's Java runtime environment version 1.5.0_03. The service replicas used JGroups 2.2.9.1 for communication, with a protocol stack configured to use TCP connections and TOTAL ordering.

Results

Figure 4.11 shows the average time per invocation of a simple no-op method that is executed either as a read-only or a modifying method. The FT*flex* infrastructure does not distributed the read-only method invocations via the group-communication framework, but instead sends them directly to one of the replicas. The figure shows the average time per invocation as measured at the client side, obtained from ten runs with 5000 client invocations each. The invocation cost for modifying methods is dominated by the cost of the totally ordered group communication. For read-only invocations, the invocation time does not depend on the number of replicas. This simple example underlines the benefit that can be achieved with the use of semantic knowledge about object methods.

4.7 Summary

Replication with Fragmented Objects

The FT*flex* architecture uses the fragmented-object model of Aspectix to provide fault-tolerant replication support. Custom fragment code can be generated automatically on a per-object basis using interface definitions, functional object implementations, and semantic annotations.

Automated Code Generation

The code generation process uses the IDL*flex* code generation tool of Aspectix. This tool was extended to support semantic annotations that influence the code generation process. An additional code transformer allows the interception of Java synchronisation mechanisms. Outside the scope of this thesis, a more flexible *Aspectix Development Kit* (ADK) is currently being developed that extends the possibilities of the code generation process. For example, languages other than Java will be supported, and the composability of multiple transformation processes is being studied. This thesis can be used to define important use-cases, which help to design the new ADK.

Supporting Object Development

Besides the basic code generation process, several mechanisms are incorporated in the FT*flex* architecture to simplify the development of replicated services. Core Aspectix mechanisms for state transfer can be used directly to automate state transfer between replicas. Code transformation is used to intercept synchronisation operations, which are then handled by protocols for deterministic multithreading in replicas. Similarly, the code transformation tool could also be used to intercept other nondeterministic operations. The current prototype implementation does not handle such nondeterminism, but it could easily be extended.

Management of Groups

Aspectix provides a flexible platform for distributed resource management, which can be used for many complex management tasks. The details of this architecture are outside the scope of this thesis. To simplify the management architecture, a replicated object provides basic functionality for self-management. This approach eliminates the need for fault-tolerant replication management, which other fault-tolerant object platforms require.

Runtime Reconfiguration

At the fragmented-object level, the basic support for dynamic reconfiguration is directly provided by Aspectix, for example by its ability to replace fragment implementations at runtime and its support for dynamic, time-bound IOR updates. A special management interface that can be used to trigger reconfigurations is provided for client applications.

Chapter 5

Deterministic Multithreading in Replicated Objects

Purpose Thread scheduling in object replicas is a source of nondeterminism. The purpose of this chapter is to describe the support for deterministic multithreaded execution of object methods in the FT*flex* architecture. The primary aim of the multithreading support in FT*flex* is to increase the performance of replicated objects and to simplify the re-use of existing object implementations for replication.

Overview Exact definitions of the synchronisation model and thread model provide the basis for a detailed discussion of deterministic scheduling algorithms. This thesis proposes replica code transformation as a novel approach to intercept synchronisation statements and to delegate them to a scheduler module. This module is called the **A**spectix **DE**terministic **T**hread **S**cheduler (ADETS). The FT*flex* architecture provides four different variants of the ADETS module.

5.1 The FT*flex* Approach to Deterministic Multithreading

Synchronisation Model FT*flex* uses a synchronisation model that supports reentrant mutex locks, condition variables, and time bounds on wait operations. The use of such a model results in a flexibility that is superior to other existing systems, which, in general, support only synchronisation with binary mutexes. The synchronisation model of FT*flex* is comprehensive enough to fully support the native synchronisation mechanisms of the Java programming language. This advantage allows the developer to use a rich set of synchronisation methods and simplifies the reuse of existing code.

Code Transformation The replication infrastructure has to intercept synchronisation statements of the replicated object in order to support deterministic multithreading. This thesis proposes code transformation as a novel approach for intercepting native Java synchronisation. The intercepted statements are then delegated to an instance of the ADETS module. The transformation approach works without low-level modifications to the Java virtual machine (JVM) or operating system. By contrast, existing systems intercept synchronisation at the operating system level (by internal modifications to the operating system or by intercepting interactions with low-level thread libraries) or with a modified JVM.

ADETS-SAT Algorithm

The first variant of the ADETS module is ADETS-SAT, which uses a **s**ingle **a**ctive **t**hread model. It is based on prior work of Jimenez-Peris et al. [JPPMA00] and Zhao et al. [ZMMS05]. These existing systems assume a simple synchronisation model that uses only binary mutexes. This thesis provides extensions for reentrant locks, condition variables, and time bounds on wait operations.

ADETS-MAT Algorithm

ADETS-MAT is a novel algorithm that supports the concurrent execution of **m**ultiple **a**ctive **t**hreads, and thus allows more concurrency than ADETS-SAT. Synchronisation is made deterministic by restricting some scheduling-related operations to a deterministically chosen primary thread. Other threads may perform local computations concurrently as long as they do not interfere with synchronisation. Such an approach has not been used previously by other systems. As in the ADETS-SAT algorithm, deterministic timeouts use communication between replicas, whereas all other synchronisation operations on mutexes and on condition variables require no communication at all.

ADETS-LSA Algorithm

The third strategy, ADETS-LSA, uses a leader-follower model inspired by the loose synchronisation algorithm (LSA) of Basile et al. [BWKI02]. Modifications to the original algorithm eliminate the dependency on global IDs for all mutexes. In Java, any object can be used as mutex, and there are no globally consistent IDs for object instances in multiple replicas. Furthermore, our ADETS-LSA algorithm provides extensions for reentrant locks, condition variables, and time bounds on wait operations.

ADETS-PDS Algorithm

The ADETS-PDS variant of the scheduling module is based on the PDS algorithm of Basile et al. [BKI03]. PDS assumes that a known set of threads executes in rounds. In each round, a thread may acquire only one (or, in a modified version, two) locks; a new round starts as soon as all threads have suspended. Similarly to the ADETS-SAT and ADETS-LSA variants, the ADETS-PDS implementation extends the original PDS algorithm with support for reentrant locks, condition variables, and timeouts.

Relevance

Of all previously published algorithms, only LSA and PDS offer the possibility to execute multiple threads concurrently in replicated objects. This thesis provides extended variants of both algorithms; these variants completely support all native synchronisation mechanisms of the Java programming language. In addition, this thesis presents the novel ADETS-MAT algorithm and shows that there are situations in which it outperforms both LSA and PDS. Hence, it provides a significant contribution to efficient active object replication in distributed systems.

5.2 Basic Assumptions

Correct Synchronisation and Deterministic Replica Implementation

A deterministic multithreaded execution of object methods is only possible if two conditions are met: the access to object state in the replica must by coordinated correctly and the replica implementation must itself be deterministic. Without correct synchronisation, the object implementation will not even work correctly in a non-replicated multithreaded execution. Without deterministic behaviour of the object implementation, a thread scheduling strategy cannot achieve determinism.

5.2.1 Synchronisation Model

Coordination of Shared Data Access

If multithreading is used, both replicated and non-replicated objects must coordinate concurrent access to shared object data. If the implementation of a non-replicated object is designed for single-threaded execution, it will not contain synchronisation statements; consequently, it can be used only for a single-threaded replicated object. The purpose of the ADETS schedulers is to allow the replication of existing servant implementations that have been designed for a multithreaded execution model, without requiring the developer to re-implement synchronisation mechanisms in the servant code. Multithreaded execution requires a correct synchronisation of the existing implementation with explicit mutex locks.

Atomic Variables

Some basic data types of Java (such as integer variables) can be accessed from multiple threads without using explicit synchronisation. The Java runtime environment guarantees that any access to such a variable is atomic. If, for example, two threads write a value to an integer variable without coordination, the variable will finally contain one of the two values. The ADETS scheduler does not support such uncoordinated variable access, and instead requires explicit synchronisation of all access to shared state.

Wait-Free Synchronisation

Using locks is not the only way to coordinate concurrent access to shared variables. In the past decade, *non-blocking* and *wait-free algorithms* have attracted much research interest [FHS04]. Such algorithms are popular especially for real-time systems, as they can avoid the priority inversion problem and have less overhead than mutex synchronisation. However, they require the use of special atomic processor instructions such as CAS (compare and swap), and are difficult to implement [MS96]. This thesis assumes traditional thread synchronisation with mutexes and does not support wait-free synchronisation.

Basic Java Synchronisation

This thesis focuses on servant objects that are implemented in the Java programming language; most of the presented concepts, however, can similarly be applied to other object-oriented languages. Java provides native mechanisms for thread synchronisation, which include binary mutex locks and condition variables [GJSB05]. Other languages such as C++ require the use of external libraries such as the POSIX thread library (pthreads) [KSS96]. For the application developer, the Java approach simplifies the development of multithreaded applications, as synchronisation mechanisms are directly included in the language syntax.

Custom Java Synchronisation

Java also permits the use of additional libraries with custom synchronisation code. A set of custom synchronisation mechanisms has been included in Java JDK 5.0 in the `java.util.concurrent` package [GJSB05]. The FT*flex* prototype implementation currently assumes traditional Java synchronisation mechanisms as described in this section. Two possibilities exist to support other custom synchronisation libraries as well. First, if the library internally uses basic low-level Java synchronisation mechanisms, these mechanisms can be intercepted. Second, the code generation tool of FT*flex* can be modified to intercept calls to external synchronisation libraries and forward these calls to an extended ADETS implementation that provides an equivalent implementation of the synchronisation methods of the library.

Reentrant Mutexes

Java associates a *binary mutex* with each object. All mutexes are *reentrant*: a single thread may acquire the same mutex lock multiple times. The lock is released only if the number of unlock operations is equal to the number of lock operations. The ADETS modules presented in this thesis use such a reentrant mutex model.

Condition Variables

Condition variables provide a mechanism that allows a thread to wait for a condition to become true inside a critical region protected by a mutex. If a thread waits on a condition variable, it releases the associated mutex lock and suspends. Another thread can signal the suspended thread to resume; the resuming thread implicitly re-acquires the mutex lock. The traditional monitor concept of Hoare [Hoa74] allows an arbitrary set of condition variables to be used within a monitor. The Java programming language provides a simplified form, in which any object can be used as mutex and as condition variable. This approach establishes a one-to-one relationship between mutexes and condition variables.

Time-Bounded Operations

In addition, Java allows the developer to specify a timeout value for operations that wait on condition variables. After the timeout expires, the thread that was suspended in a waiting operation resumes without having been notified. As condition variables are an important mechanism for synchronisation, they are supported in the synchronisation model of this thesis.

5.2.2 Details of Java Synchronisation

Java Synchronisation

The Java programming language provides the `synchronized` keyword to specify synchronisation operations on a mutex. Every instance of a Java object can be used as a mutex. The `synchronized` keyword can be used as a method modifier or as a modifier of a code block within a method.

Synchronised Instance Methods

- A *synchronised instance method* of an object will lock the mutex of the object at method entry and unlock it when leaving.

Synchronised Static Methods

- A *synchronised static method* of a class will lock the mutex of the class meta object at method entry and unlock it when leaving.

Synchronised Blocks

- A block within a method preceded by the `synchronized` keyword (or, briefly, a *synchronised block*) explicitly specifies an object that will be locked at the beginning of the block and released at the end of the block.

Semantics of Synchronised Elements

A `synchronized` element implicitly specifies a lock and an unlock operation. Such a structure is less error prone than explicit locking statements; a mutex can neither be unlocked without having been locked before, nor can the unlock be "forgotten". An execution thread can pass through multiple nested `synchronized` elements, which means that multiple mutexes are locked successively. The implicit lock acquisition is less flexible than explicit lock and unlock operations, as all locks have to be unlocked by the same thread and in reverse order of lock acquisition.

Condition Variables

In addition to a mutex, each object in Java provides a condition variable. For this purpose, all objects inherit the final methods `wait()`, `notify()`, and `notifyAll()` from the core `java.lang.Object` class. These methods may be called only after obtaining the mutex of the object. The `wait()` method releases the mutex and blocks until it is woken up either by a notification or a timeout. The `notify()` method signals one out of all threads blocked in a `wait()` operation to resume. The `notifyAll()` method unblocks all waiting threads. Typically, `wait()` is used in a loop that checks a condition.

Nondeterminism

Multithreaded Java programs face several sources of nondeterminism related to thread management:

a) The order in which parallel threads are scheduled is unpredictable. If two threads concurrently access the same data and consequently request the corresponding mutex, the order in which the mutex is granted is nondeterministic.

b) If a thread wants to acquire a mutex that is held by another thread, it is suspended until the mutex becomes available. If multiple threads are suspended, the selection of the thread to which the mutex is granted next after an unlock operation is nondeterministic.

c) In both notification operations (`notify()` and `notifyAll()`), one cannot predict or specify the order in which waiting threads wake up and execute, as such an order is not defined by the Java language specification.

Challenges

Replicating an existing servant implementation that uses the native synchronisation mechanisms of Java should not force the developer to change the synchronisation code. This requirement makes it necessary for the fault-tolerance infrastructure to remove all sources of nondeterminism that can arise from the execution of concurrent threads. The order in which locks are granted and waiting threads are resumed has to be made deterministic.

5.2.3 Replica Determinism

Determinism

Deterministic replica behaviour is obtained only if the replica implementation provides some form of predictability. For a single-threaded execution model, it is easy to define a *determinism requirement* that the implementation must fulfil.

Definition 5.1 (Determinism in a Single-Threaded Model) *Let $S(t)$ be the state of the object at time t, and r a client request. A replica implementation is deterministic if the request r and the object state $S(t_a)$ at the beginning of the execution of r uniquely define the object state $S(t_b)$ at the end of the execution of r.*

Problem of Multithreading

In a multithreaded execution, the concurrent execution of threads is a potential source of nondeterminism. The replica implementation cannot provide determinism independently of the scheduling strategy. The preceding definition of determinism is not useful for a multi-threaded model: during the execution of request r, other threads can concurrently modify shared state data. Such modifications invalidate the assumption that the state $S(t)$ has a unique value at the end of the execution of r.

Data Model

A definition of determinism that permits multithreaded execution must be based on a more fine-grained observation of the object state. The ADETS scheduler assumes that all access to shared data is protected by mutexes. In addition, a thread may access local data that is not accessible for other threads. Given a specific set of locks, a thread will only modify (a) local data and (b) shared data that is protected by the locks. These modifications have to be deterministic; no other thread may modify this part of the object state concurrently.

Formal Model

This assumption can be defined in a more formal way. For this purpose, the execution of a request is modelled as a sequence of *thread execution intervals* (see Definition 5.2). *Piecewise thread determinism* makes assumptions about the thread behaviour during such an interval (see Definition 5.3). If the replica implementation is piecewise deterministic and if the infrastructure provides deterministic thread scheduling, then the replica behaviour will be deterministic.

Thread Execution Intervals

Definition 5.2 (Thread Execution Intervals) *A scheduling point s_i of a thread T is defined by any of the following activities of T: thread creation, request and release of a mutex lock, wait request on a condition variable, nested invocation, and thread termination. An execution interval e_i of a thread is the activity of a thread between s_i and s_{i+1}.*

Scheduling Points

Thread creation always defines the first scheduling point s_0, and thread termination defines that last scheduling point s_N. The scheduling points $s_k, 0 < k < N$ may temporarily suspend the thread waiting for a mutex, for a condition variable, or for a nested invocation reply. In these cases, the next execution interval e_k is started as soon as the lock is granted, the wait operation is notified or times out, or the reply for the nested invocation arrives, respectively.

During an execution interval e_i, the set of mutexes that the thread has locked does not change. Any operation that changes the set of locked mutexes starts a new execution interval. A nested invocation does not change the lock set. It is, however, a potential source of nondeterminism that is outside the control of the replica implementation. For this reason, nested invocations also start a new execution interval.

Piecewise Thread Determinism

Definition 5.3 (Piecewise Thread Determinism) *Let $L_T(t)$ be the local state of a thread T at time t, and $S_{T,i}(t)$ be the part of the shared object state that thread T can access in execution interval e_i on the basis of previous lock operations. A thread T is piecewise deterministic iff the local state $L_T(t_a)$ and the protected part of the shared state $S_{T,i}(t_a)$ at the beginning of e_i uniquely define the state of $L_T(t_b)$ and $S_{T,i}(t_b)$ at the end of the execution interval e_i.*

The developer of the replicated object has to make sure that the replica implementation is piecewise deterministic and that during an execution interval a thread reads or modifies only those parts of the shared object state that are not concurrently modified by other threads.

5.2.4 Granularity of Synchronisation

Synchronisation per Object

In the following it is assumed that each instance of a replica group is independent from other instances. Each instance accesses only internal data directly, and interacts with other instances via remote invocations; each replica group has its own totally ordered group communication facility to receive client requests and replies from nested invocations. In this model, the unit of thread synchronisation is the replica group instance. Internally, each replica uses a separate instance of an ADETS module.

Nested Invocations

A replica group A may invoke remote methods of an independent replica group B. In this case, the infrastructure makes sure that a single invocation is made from A to B. This means that all replicas of A have to be coordinated to make a joint invocation of a method at B. The result of this invocation is then propagated to all replicas of A, as described in Section 4.2.

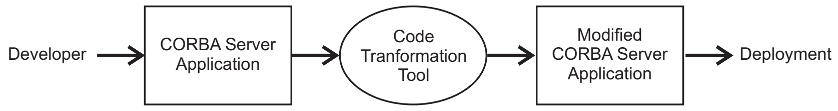


Figure 5.1: Software transformation process for synchronisation transparency

Joint Scheduling for Multiple Objects

It is possible to combine multiple objects within one replication group; in this case, these objects can have access to common shared data. A nested invocation from one object group to the other group can be implemented by local method invocations. In this scenario, all objects of a single replication group must use common thread synchronisation, which requires common group communication and a common ADETS instance. This variant requires no modification of the scheduling algorithms themselves.

5.3 Interception of Synchronisation

Interception is Needed

Using multiple threads in replicated objects requires that the replication infrastructure be able to control concurrent actions. This fact applies to all approaches to deterministic multithreading. For this purpose, the infrastructure must control the synchronisation actions that determine the order in which shared state is accessed. This means that the infrastructure must *intercept* all synchronisation-related statements in the replica code.

Interception Approaches

Interception can be implemented using low-level approaches, such as using specifically modified hardware, operating systems, and Java virtual machines. In contrast, FT*flex* aims at supporting deterministic multithreading without such low-level means. In programming languages like C++, it is possible to redirect local library calls to, e.g., the POSIX thread library [KSS96]. For example, the Eternal system [NMMS99] uses this approach for transparent interception. In Java, the thread synchronisation primitives are directly integrated in the programming language [GJSB05], which makes this approach less feasible.

Source-Code Transformation

As an alternative, this thesis proposes a code-transformation approach. A software transformation tool of the FT*flex* middleware converts native Java synchronisation primitives into synchronisation calls that interact with the ADETS module [RKDH06]. This approach is transparent to the application developer, as the application can use all native Java primitives for synchronisation, without consideration for the transformation process. Before application deployment, the code is passed through the code transformation tool (Figure 5.1).

The synchronisation methods that need to be intercepted include all operations on mutexes and condition variables. The code transformer of FT*flex* replaces these operations with method invocations at the ADETS scheduler. The interface of the ADETS module is shown in Figure 5.2; plug-in components provide implementations of this module.

Mutex Access

All `synchronized` elements in the source code need to be replaced by a pair of `lock()/unlock()` invocations at the ADETS module. A `try/finally` construct is used to make sure that `unlock()` is always called, even if the method prematurely exits via an exception or a return statement. For a *synchronised instance method*, `this` is passed as mutex reference to the lock and unlock

```
interface ADETS {
    void lock(Object o);
    void unlock(Object o);

    void swait(Object o) throws InterruptedException;
    void swait(Object o, long timeout) throws InterruptedException;
    void swait(Object o, long timeout, int nanos) throws InterruptedException
    void snotify (Object o);
    void snotifyAll (Object o);

    Reply invokeNested(Request r);

    void start ();
    void stop ();
    void getState(ObjectOutputStream oos);
    void setState(ObjectInputStream ois);
}
```

Figure 5.2: Generic interface of the ADETS module

statements. For *synchronised static methods*, a reference to the class meta object (i.e., `<classname>.class`) has to be passed instead. *Synchronised blocks* are more complicated, as a reference to a custom variable can be passed as mutex object and this reference can be changed within the synchronised block (e.g., `synchronized(x){x=y;}`). Consequently, a copy of the mutex reference must be made in order to be able to call `unlock()` with the right reference at the end of the block.

Example The example in Figure 5.3 illustrates the effect of the code transformation. A synchronised method is converted to a `lock()` call at the beginning and an `unlock()` call at the end of the method, passing the `this` reference as lock object.

Condition Variables In addition to mutex synchronisation, the ADETS module also provides methods for condition variables. Calls to the native `wait()`, `notify()` and `notifyAll()` methods are simply transformed to corresponding calls to the scheduler instance of the replica. As these methods have reserved names that cannot be used in custom object implementations, the Java implementation of the ADETS interface uses the prefix “`s`” for the corresponding methods. For simplicity, the descriptions of the ADETS algorithms in the next sections do not use this prefix.

Nested Invocations The ADETS scheduler also needs to be aware of nested invocations. These invocations are potential scheduling points (e.g., in a single-active-thread model), and the delivery of the invocation reply has to be coordinated by the scheduler. For this purpose, an access fragment that handles a nested invocation calls `invokeNested` at the scheduler. This method (a) informs the scheduler about the invocations, (b) executes the actual nested invocation, and (c) lets the scheduler decide when to resume to callee by returning the reply.

State Transfer The ADETS module has to be integrated in the state-transfer mechanism that is used to integrate new replicas. Therefore, the state transfer has to be coordinated with the execution of client requests, and the scheduler state itself needs to be transferred. The ADETS interface provides two methods,

```
public class Queue extends ... {
    public synchronized
    void remove()
    {
        while(data.size()==0)
            wait();
        return data.remove(0);
    }



    public synchronized
    void append(String x)
    {
        data.add(x);
        notify();
    }



}
```

⇒

```
public class Queue extends ... {
    public void remove() {
        scheduler.lock(this);
        try {
            while(data.size()==0)
                scheduler.swait(this);
            return data.remove(0);
        } finally {
            scheduler.unlock(this);
        }
    }
    public void append(String x) {
        scheduler.lock(this);
        try {
            data.add(x);
            scheduler.snotify(this);
        } finally {
            scheduler.unlock(this);
        }
    }
}
```

Figure 5.3: Example of code transformation for Java mutexes and condition variables

`stop()` and `start()`, that are called by the infrastructure before and after the state transfer. A `stop()` operation signals the ADETS implementation to stop processing new requests; the operation blocks until all requests that are still being processed have finished. After that, a consistent state transfer without concurrent execution of other replica methods can be performed. The state transfer not only transfers the state of the replica implementation, but also includes the state of the ADETS instance, which is accessible via the methods `getState()` and `setState()`.

5.4 ADETS-SAT: A Single Active Thread Algorithm

ADETS-SAT: Basics

This section defines the non-preemptive ADETS-SAT algorithm [DHRK06], which uses a single-active-thread approach. The algorithm is inspired by previous works of Zhao et al. [ZMMS05] and Jimenez-Peris et al. [JPPMA00] (see Section 3.2). The algorithm does not allow true concurrency; a new thread is allowed to start only if the execution of a request is suspended due to a nested invocation or an unavailable lock. ADETS-SAT is the simplest variant of the ADETS algorithms defined by this thesis.

LockedMap:	Map<Object,[ThreadID,count]>
MutexWaitMap:	Map<Object, Queue<ThreadID>>
CondWaitMap:	Map<Object, Queue<[ThreadID,UUID]>>
idle :	boolean

Figure 5.4: Basic data structures of the ADETS-SAT algorithm

5.4.1 Thread States

Thread States

Threads executing methods of a replica group can be in one of the following states: *runnable*, *suspended*, or *terminated*.

- A thread is *terminated* if it has stopped executing and will never resume. A terminated thread is later cleaned up by the garbage collector.
- A thread is *suspended* if it is (a) waiting for a new request, (b) waiting for a mutex lock, (c) waiting on a condition variable, or (d) waiting for the reply of a nested invocation.
- A thread is *runnable* if it is neither terminated nor suspended.

Only a Single Runnable Thread

The ADETS-SAT algorithm makes sure that only one deterministically chosen thread is in the state *runnable*. The algorithm is non-preemptive, and no explicit *ready* state is used. Instead, a new thread created or moved from *suspended* to *runnable* state only after the currently *runnable* thread terminates or suspends. The decision about which thread to resume or create is fully deterministic under the control of the ADETS-SAT scheduler. This behaviour is identical to the original algorithm of Zhao et al [ZMMS05].

5.4.2 Data Structures

Data Structures

Figure 5.4 shows the basic data structures used by the ADETS-SAT algorithm. The term `Object` is used to refer both to a mutex and to a condition variable; this implies the assumption that for each mutex there exists exactly one condition variable[1].

LockedMap

`LockedMap` maps object references to threads that hold the mutex lock of the object. As the algorithm simulates the reentrant behaviour of Java monitors, a single thread may lock the mutex multiple times. The lock count of the mutex is stored in the map together with the thread ID. On unlock operations, the lock count is decremented, and when it reaches zero the object is removed from `LockedMap`.

MutexWaitMap

`MutexWaitMap` maps object references to an ordered list of threads that want to acquire the mutex lock of the object. Threads are added to `MutexWaitMap` if they try to lock a mutex that is currently held by another thread (indicated by an entry in `LockedMap`). Threads are also added to `MutexWaitMap` if they have been suspended by a `wait()` call and subsequently are resumed by a `notify()` operation or timeout.

CondWaitMap

`CondWaitMap` maps object references to an ordered list of threads that use

[1]This assumption simplifies the description of the algorithm and is justified for the Java-based environment, which provides such an one-to-one relation; the algorithm can be extended to a more general condition variable concept by providing a mapping function from condition variables to mutexes.

this object to wait on a condition variable. A unique ID of the wait operation is stored in the map together with the thread ID. The unique ID is required to unambiguously map timeout messages to waiting threads.

idle

`idle` is set to true if no thread is currently runnable. If a new message arrives and `idle` is true, then the scheduler is triggered to use the new message to create or resume a thread.

5.4.3 Description of the ADETS-SAT Algorithm

Pseudocode of the Algorithm

Figure 5.5 shows a pseudocode description of the ADETS-SAT algorithm. The algorithm assumes that intercepted synchronisation statements of the servant implementation are delegated to the corresponding scheduler methods. The code provides methods for `lock()` and `unlock()` operations on reentrant mutexes as well as for `wait()`, `notify()`, and `notifyAll()` operations on condition variables. Furthermore, it contains functionality for interrupting `wait()` operations by time bounds.

The `schedule()` Method

The internal `schedule()` method (lines 1–21) is used to create or resume the next active thread. A call to `schedule()` is only made when the currently running thread blocks or terminates. A thread may block at a `lock()` operation (line 47), at a `wait()` operation (line 58), and when it makes a nested invocation (line 35); in addition, termination is handled in line 32. The `schedule()` method also calls itself recursively after processing a `Timeout` message. Furthermore, if `schedule()` terminates due to an empty message queue (line 8), it sets `idle` to *true* and is called again as soon as the next message arrives (line 29).

New Active Thread Selection

The `schedule()` method deterministically chooses the next thread to be created or resumed. First, it checks whether a thread can be resumed without processing any incoming messages (lines 2–6). This may happen if a thread waiting on a `lock()` or `wait()` operation can continue due to a previous `unlock()`, `notify()`, or timeout. In this case, there is an object entry in `MutexWaitMap` that is not in `LockedMap`.

Processing Incoming Messages

Otherwise, the first message from the incoming message queue is processed. If no such message exists, `schedule()` exits, and the system remains idle until the next message arrives, which causes `schedule()` to be called again. All messages arrive at all replicas in total order. Messages can be new invocations, replies from nested invocations, and `Timeout` messages that are used internally to implement time-bounded wait operations.

Variants of the Thread Model

The choice of the next message to process depends on the thread model in use; common models are *thread-per-request*, *thread-per-client*, *thread-per-transaction*, and *thread-pool* [ZMMS05]. In a thread-per-request model, with an unbounded number of threads, the message at the head of the incoming queue can always be processed by creating a new thread if no existing thread handles this message. In all other models, it can happen that a request message cannot be processed, because a new thread for handling that request currently cannot be created, or because it needs to be handled by an existing thread that is currently suspended (waiting on a lock, condition variable, or nested invocation reply). In that case, the message queue has to be implemented as a priority queue that defers currently unprocessable requests.

Lock Operations

An intercepted `lock()` operation checks if the requested mutex is either available or locked by the current thread. In these cases, the mutex is successfully granted to the thread or the reentrance count is increased, respectively,

```
function schedule():
    find obj in keys(MutexWaitMap)\keys(LockedMap)
    if obj exists:
        tid := MutexWaitMap(obj).removeFirst()
        LockedMap(obj) := [tid,1]
        tid.resume()
        return
    if inQueue.isEmpty():
        idle := true;
        return
    msg := inQueue.removeFirst()
    if msg is CLIENT_REQUEST:
        start new request handler thread
    if msg is TIMEOUT(obj,tid,uuid):
        if CondWaitMap(obj).contains([tid,uuid]):
            CondWaitMap(obj).remove([tid,uuid])
            MutexWaitMap(obj).append(tid)
        schedule()
    if msg is NESTED_REPLY(tid,value):
        tid.deliver(value)
        tid.resume()

function receive(message):
    if message is TIMEOUT(obj,tid,uuid)
        Timer.cancel(TIMEOUT(obj,tid,uuid))
    inQueue.append(message)
    if idle == true:
        idle := false
        schedule()

On termination of thread tid:
    schedule()

function invokeNested(request) of thread tid:
    schedule()
    request.invoke();
    tid.suspend()
    return tid.getDelivered()

// intercepted synchronisation functions
function lock(obj) called by thread tid:
    [locktid, i] := LockedMap(obj)
    if locktid == nil:         LockedMap(obj) := [tid,1]
    else if locktid == tid:    LockedMap(obj) := [tid,i+1]
    else:
        MutexWaitMap(obj).append(tid)
        schedule()
        tid.suspend()
```

Figure 5.5: The ADETS-SAT algorithm

```
49 function unlock(obj) called by thread tid:
50     [tid', i] := LockedMap(obj); assert tid'==tid
51     if i==1: LockedLockedMap.remove(obj)
52     else      LockedMap(obj) := [tid,i−1]
53
54 function wait(obj, timeout) called by thread tid:
55     [tid,n] := LockedMap.remove(obj) // fully release lock
56     uuid := new unique ID
57     CondWaitMap(obj).append([tid, uuid])
58     schedule()
59     if timeout > 0:
60         Timer.setup(timeout, [obj, tid, uuid])
61     tid.suspend(); // until moved to LockedMap by schedule
62     LockedMap(obj) := [tid,n]
63
64 function notify(obj) called by thread tid:
65     if CondWaitMap(obj) ≠ nil:
66         [tid,uuid] := CondWaitMap(obj).removeFirst()
67         Timer.cancel([obj, tid, uuid])
68         MutexWaitMap(obj).append(tid)
69
70 function notifyAll(obj) called by thread tid:
71     for all elements [$tid_i$, $uuid_i$] in CondWaitMap(obj)
72         Timer.cancel(obj,$tid_i$,$uuid_i$)
73         MutexWaitMap(obj).append($tid_i$)
74     CondWaitMap.delete(obj)
75
76 function Timer.setup(timeout, message):
77     Schedule sending message via abcast after timeout ms
78
79 function Timer.cancel(message):
80     Cancel sending message if not yet sent
```

and the thread can continue. Otherwise, the lock is unavailable and the thread is suspended.

Unlock Operations

Unlocking a mutex causes only a local modification to `LockedMap` (lines 49–52); other threads that might have been waiting for the released lock are not resumed immediately. They are resumed later in `schedule()`, which is invoked as soon as the current thread terminates or suspends.

Operations on Condition Variables

An intercepted `wait()` operation first releases the corresponding mutex, and then suspends. After the suspended thread is resumed, it has to re-acquire the mutex lock with the same reentrance count as the mutex had before the `wait()` operation. A waiting thread can be resumed by a `notify()` or `notifyAll()` operation, or by a timeout.

Handling Timeouts

If the replicated application issues a time-bounded `wait()` operation, the emission of a `Timeout` message is scheduled (line 60). The message is sent to all group members via totally ordered group communication. A call to `schedule()`, which processes `Timeout` messages in the same way as client requests or nested invocation replies, is only made if the active thread suspends or terminates. If a `notify()` or `notifyAll()` operation is called before a `Timeout` message arrives, the unique ID of the `wait()` operation is removed from `CondWaitMap` (lines 66 and 74), and the `Timeout` message has no effect (line 15). The first reception of a `Timeout` message with a specific ID cancels the emission of a `Timeout` message with the same ID, if such a message has not yet been sent. If the timeout expires locally in multiple nodes, no guarantee is made that no duplicated messages are sent. At reception time, the unique ID makes sure that only the first `Timeout` message has an effect, while duplicated messages with the same ID are ignored.

5.4.4 Verification of the ADETS-SAT Algorithm

Lemma 5.1: Deterministic strategy of `schedule()`

The following Lemma 5.1 shows that in a suspended state, the `schedule()` method uses a deterministic strategy to select a new thread and to modify the internal scheduling data structures. This lemma is subsequently used to show that the scheduler, together with piecewise determinism of the replica implementation, maintains replica consistency.

Lemma 5.1 *Given a state S of the synchronisation data structures and a totally ordered sequence M of incoming messages at the head of the message queue, the thread that is resumed next and the modifications to the synchronisation data structures are deterministically defined by S and M.*

Verification of Lemma 5.1:

No Concurrency at `schedule()` Invocation

(1) The `schedule()` function is called only if (a) no thread is active and a new message arrives, (b) the active thread terminates, (c) the active thread issues a nested invocation, (d) the active threads suspends in a `lock()` operation because the requested lock is held by another thread, and (e) the active thread suspends in a `wait()` operation on a condition variable. In all these cases, no other thread is active concurrently with `schedule()`.

Finding a `MutexWaitMap` Entry not in LockedMap.

(2) The first step that `schedule()` takes is to check whether there is any object that has an entry in `MutexWaitMap`, but no corresponding entry in `LockedMap`. An entry can appear in `MutexWaitMap` for two reasons. Either a thread invoked `lock()` on an unavailable mutex and put itself into the `MutexWaitMap` before suspending. Or a thread invoked `wait()` and a later `notify()`, `notifyAll()`, or `Timeout` message caused the waiting thread to continue, forcing it to re-acquire the previously held mutex lock. Due to (1), no such action can occur concurrently with the `schedule()` execution. Given that finding an object in `MutexWaitMap` that is not in `LockedMap` is implemented with a deterministic strategy, the selection of the thread to resume and the corresponding modification of the synchronisation data structures are deterministic.

Processing Next Incoming Message

(3) If no thread is resumed in (2), the next message from the incoming message queue is processed. The message queue can contain client requests, nested invocation replies, and timeout messages. If the message was a nested invocation reply or a request that is to be handled by an existing thread, this thread is resumed. If the message is a new client requests, a new thread is created and started. If it is a `Timeout` message, it causes a deterministic modification of the synchronisation data structures and a recursive call to `schedule`.

(4) If no message is present in the incoming message queue, `schedule()` terminates and is called again as soon as the first message arrives. Because of this, the effect is the same as if the message had been available already in the first `schedule()` execution. *No Incoming Message*

Lemma 5.2 *Given a consistent initial state, an identical sequence of messages arriving at all replicas, and piecewise deterministic behaviour of the replica implementations, the ADETS-SAT algorithm will maintain consistency of all replicas.* *Lemma 5.2: ADETS-SAT Maintains Replica Consistency*

Verification of Lemma 5.2:

(1) By assumption, the initial state of all replicas is identical, and no thread is running. The first arriving message will trigger a `schedule()` call. *Initial State*

(2) Again by assumption, a single active thread will have piecewise deterministic behaviour. That is, it will execute the same sequence of operations (state modifications, lock requests and releases, condition variable notification, and nested invocation) in all replicas until it finally terminates or suspends (in a lock operation, in a wait operation, in a nested invocation, or when waiting for a new request). *Piecewise Deterministic Behaviour*

(3) After the active thread terminates or suspends, a call to `schedule()` is triggered. By Lemma 5.1, this call will make deterministic modifications to the synchronisation data structures and will make a consistent selection of the next active thread. *Call to `schedule()`*

In other words, the execution of a replica implementation with an ADETS-SAT instance is an execution sequence of (a) scheduling operations implemented by the `schedule()` function of ADETS-SAT and (b) piecewise deterministic execution intervals. The determinism of (a) is guaranteed by the ADETS-SAT algorithm. The determinism of (b) has to be provided by the replica developer. *Conclusion*

5.5 ADETS-MAT: A Multiple Active Thread Algorithm

Unlike the previously discussed ADETS-SAT variant, the ADETS-MAT algorithm [RHD+06] belongs to the MultipleActiveThreads (MAT) category. Multiple threads can run concurrently within a single object. The discussion of ADETS-MAT starts with an informal description of the data structures and the functionality of the ADETS-MAT algorithm. A proof of correctness will be given in Section 5.5.4. *Overview*

The algorithm again requires that the implementation of a replicated object protect the access to common state variables by mutex locks, and that it be piecewise deterministic as defined in Section 5.2.3. The algorithm supports Java-style condition variables (i.e., each mutex has an associated condition variable, on which the application calls `wait()`, `notify()` and `notifyAll()` operations). Threads that are blocked in a `wait()` operation can also be unblocked by a timeout. *Assumptions*

ActivePrimary:	ThreadID
LockedMap:	Map<Object,[ThreadID,count]>
MutexWaitMap:	Map<Object, Queue<ThreadID>>
CondWaitMap:	Map<Object, Queue<[ThreadID,UUID]>>
PrimCandidates:	Queue<[ThreadID, Queue<Action>]>
CurrentActionList:	Map<ThreadID, reference to Queue<Action>>

Figure 5.6: Basic data structures of the ADETS-MAT algorithm

5.5.1 Data Structures

Figure 5.6 shows the essential data structures used by the ADETS-MAT algorithm. Identical to the ADETS-SAT algorithm, the term `Object` is used to refer both to a mutex and to a condition variable.

ActivePrimary

`ActivePrimary` specifies the currently active primary thread. This thread locally defines the order of mutex lock acquisitions. An arbitrary number of additional secondary threads can run in parallel, but these threads may not influence the lock acquisition order. The transfer of the role of being the active primary from one thread to another is defined by deterministic rules. Only the active primary thread can acquire or release locks (i.e., modify the entries in `LockedMap`) or adjust the list of threads waiting for a lock or condition variable (i.e., modify the entries in `MutexWaitMap` and `CondWaitMap`).

LockedMap

`LockedMap` is used to store the information about which mutex is locked by which thread. Reentrant locks are supported by a counter, which is incremented on each lock operation requested by a single thread, and decremented on each unlock operation of the same thread. If the counter reaches the value 0, the lock is no longer held by the thread, and the mutex entry is removed from the map. Any mutex not in `LockedMap` is free. Only the active primary thread may add a new entry to `LockedMap` or remove an entry from it.

MutexWaitMap

`MutexWaitMap` stores a list of threads that are waiting for a mutex. Only the active primary thread will be added to this map; after the addition, the primary will suspend and a new primary will be selected consistently in all replicas.

CondWaitMap

`CondWaitMap` stores all threads that are waiting on a condition variable. Identical to `MutexWaitMap`, threads will be added only while they are active primary. In addition to the thread ID, a unique ID is stored to identify the `wait()` operation. The unique ID is used to correctly assign timeout messages to `wait()` operations.

PrimCandidates

`PrimCandidates` is an ordered queue with an entry for each received message. An entry contains a reference to a secondary thread T_s that handles the message in parallel to the primary thread, and a list of deferred actions (e.g., `unlock()` and `notify()` operations) that T_s has requested, but which may only be executed after T_s becomes the primary thread. This list is called the *action list* of the thread. If T_s performs an action that is not allowed for secondary threads (`lock()` and `wait()`), the thread suspends until it becomes the active primary.

CurrentActionList

Multiple entries in `PrimCandidates` can reference the same thread. This situation arises if a thread issues multiple nested invocations while executing as a secondary thread. Each invocation reply creates a new `PrimCandidates` entry. Each entry references a separate action list to record the deferred actions that

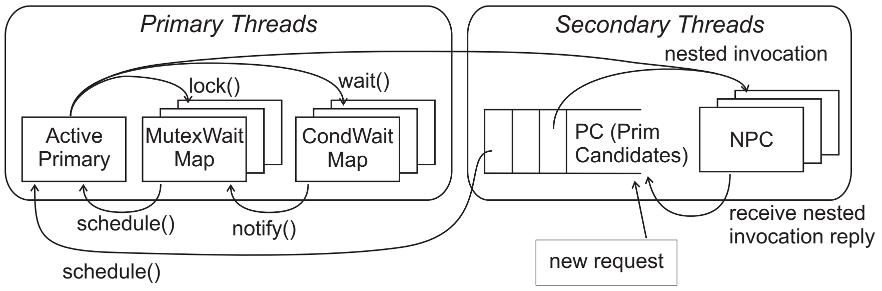


Figure 5.7: Transitions between thread states in the ADETS-MAT algorithm

the thread executed between two nested invocations. The `CurrentActionList` maps thread IDs to the deferred action list in `PrimCandidates` that corresponds to the current execution (i.e., the action list in the `PrimCandidates` entry by which the thread was last resumed).

5.5.2 Overview of the ADETS-MAT Algorithm

Thread Classes

Figure 5.7 illustrates the possible states of a thread and the possible transitions between states. A thread belongs to the set of either the *primary* or the *secondary* threads. The *primary* set consists of a single active thread (stored in `ActivePrimary`) and a set of suspended threads that are waiting for a lock or for a condition variable (stored in `MutexWaitMap` and `CondWaitMap`, respectively). *Secondary* threads that are waiting for a nested invocation reply are non-primary candidates (NPC); all other secondary threads are references from an entry in `PrimCandidates` and are called primary candidates (PC). If a primary candidate issues a nested invocation, it becomes NPC, but is still referenced by a `PrimCandidates` entry.

`PrimCandidates`

The order in the `PrimCandidates` queue is defined by the total order of messages received from group communication. Each entry in the queue has a reference to a secondary thread. When the queue head is used to select the next active primary thread, the referenced thread may still be running, it may be suspended because of an operation that only the active primary may perform, it may be suspended due to a nested invocation, or it may have terminated. If the referenced thread has terminated or has issued a nested invocation, it is not selected as new active primary; instead, the queue entry is used only to process the corresponding action list of deferred actions that the thread executed before terminating or issuing the nested invocation. The selection of the next active primary then proceeds with the next `PrimCandidates` entry.

Message Arrival

Every arriving message creates an entry in the `PrimCandidates` queue with a reference to a corresponding thread. In case of a client request, this reference points to the new thread that is created and started to handle the request. If the message is a nested invocation reply, the reference points to the thread waiting for this reply, and this thread is resumed. In case of a `Timeout` message, the thread reference points to the thread that executed the corresponding `wait()` operation, and a timeout entry with the UUID from the message is stored in the action list for processing as soon as the thread becomes primary.

schedule() Operation

A `schedule()` operation is responsible for activating a new primary thread. It is called if the current active primary terminates or suspends, or if an entry is added to an empty `PrimCandidates` queue. The method first tries to select a runnable thread from `MutexWaitMap`. If no such thread exists, the queue head of `PrimCandidates` is examined. If the queue is empty, `schedule()` terminates. Otherwise, the head element is removed from `PrimCandidates` and the referenced thread becomes active primary.

Nested Invocation

If the primary thread issues a nested invocation, it cannot resume before the corresponding reply arrives. Because of this, it is removed from the set of primary threads and becomes an NPC member. As soon as the nested invocation reply arrives, the reply message with a reference to the thread is added to `PrimCandidates`; the thread becomes a PC and resumes execution as a secondary thread.

5.5.3 Specification of the ADETS-MAT Algorithm

Pseudocode

Figure 5.8 shows a specification of the ADETS-MAT algorithm in pseudocode. The main component of the scheduler is the `schedule()` function implemented in lines 1–26. The function is called (a) when no active primary exists and a new message arrives (line 30; `appendPrimCandidates` is called from the `receive()` function), (b) when the current primary thread terminates (line 44), and (c) when it suspends (it issues a nested invocation, line 48; it calls `lock()` on a mutex locked by another thread, line 79); or it calls `wait` on a condition variable, line 89.

Examination of `MutexWaitMap`

The `schedule()` implementation first examines `MutexWaitMap` for resumable threads that are blocked on a synchronisation operation (lines 2–7). A thread can be resumed if it has requested a mutex lock that is now available (i.e., has no entry in `LockedMap`). This also covers threads that issued a `wait()` operation: if a thread in `CondWaitMap` is notified by another thread or a timeout, it is moved from `CondWaitMap` to `MutexWaitMap`, as it has to re-acquire the lock prior to continuation. If a runnable thread is found, it is resumed (line 7). To make a deterministic selection in case of multiple available threads, `MutexWaitMap` has to be an ordered map, sorted, for example, by the sequence of lock requests.

Examination of `PrimCandidates`

If no resumable thread is found, the `PrimCandidates` queue is examined. If the queue is empty, no primary thread is selected, and `schedule()` terminates (line 9); it is re-invoked as soon as a new message arrives. Otherwise, `schedule()` picks the first element from `PrimCandidates` (line 10) and processes the action list from the queue entry (i.e., it executes deferred actions, lines 11–22). If the thread that corresponds to the queue element has been blocked due to a `wait()` or `lock()` call while being secondary, it is resumed. If the thread is not runnable (it has terminated or has issued a nested invocation), `schedule()` is called again to repeat the selection of a new primary thread (line 26). Otherwise, it becomes the new primary thread.

Processing of New Messages

Lines 33–41 show the processing of new messages from the group communication system. Three kinds of messages may arrive: client requests, `Timeout` messages, and nested invocation replies. For client requests, a new thread is created. For nested invocation replies, the thread waiting for the reply is resumed. In both cases, an entry is added to `PrimCandidates` with a reference to the thread and an empty action list. For `Timeout` messages, an entry with

```
function schedule():
    find obj with MutexWaitMap(obj) $\neq$ nil and LockedMap(obj) = nil
    if obj exists:
        tid := MutexWaitMap(obj).removeFirst()
        LockedMap(obj) := [tid,1]
        ActivePrimary := tid;
        tid.resume(); return  // resume suspended thread
    if(PrimCandidates.isEmpty())
        ActivePrimary := nil; return
    [tid, alist] := PrimCandidates.removeFirst()
    foreach entry in alist:
        case TIMEOUT(uuid):
            if [tid, uuid] $\in$ CondWaitMap(obj):
                CondWaitMap(obj).remove([tid, uuid])
                MutexWaitMap(obj).append(tid)
                tid := nil
        case TERMINATE, WAIT_NESTED:
            tid := nil
        case WAIT, LOCK:
            // thread is resumed below as ActivePrimary
        case UNLOCK(obj)/NOTIFY{|ALL}(obj):
            call primary{Unlock|Notify|NotifyAll}(obj, tid)
    if(tid!=nil):
        ActivePrimary := tid
        if(tid is suspended) tid.resume()
    else schedule()

function appendPrimCandidate([tid, alist]):
    PrimCandidates.append([tid,alist])
    if(ActivePrimary==nil) schedule()
    CurActionList(tid) := pointer to alist

function receive(message):
    if message is new client request:
        tid := new thread(message)
        appendPrimCandidate([tid, ()]); tid.run()
    if message is TIMEOUT(obj,tid,uuid)
        Timer.cancel(TIMEOUT(obj,tid,uuid))
        appendPrimCandidate([tid, (TIMEOUT(uuid))])
    if message is nested invocation reply for thread tid:
        appendPrimCandidate([tid, ()]); tid.deliver(message)  // resume thread

On termination of thread tid:
     if tid == ActivePrimary: schedule()
     else CurActionList(tid).append(TERMINATE)

function invokeNested(request) by thread tid:
    if ActivePrimary == tid: schedule()
    if ActivePrimary $\neq$ tid: CurActionList(tid).append(WAIT_NESTED)
    request.invoke()
    tid.suspend()  // until reply is received
    return tid.getDelivered()
```

Figure 5.8: The ADETS-MAT algorithm

```
function **primaryUnlock**(obj, tid):
    [tid‘, k] := LockedMap(obj); assert tid‘ == tid
    if k>1: LockedMap(obj) := [tid, k−1]
    else:   LockedMap.remove(obj)

function **primaryNotify**(obj):
    [tid, uuid] := CondWaitMap(obj).removeFirst()
    Timer.cancel(TIMEOUT(obj,tid,uuid))
    MutexWaitMap(obj).append(tid)

function **primaryNotifyAll**(obj):
    for all elements [$tid_i$, $uuid_i$] in CondWaitMap(obj):
        Timer.cancel(TIMEOUT(obj,$tid_i$,$uuid_i$))
        MutexWaitMap(obj).append($tid_i$)
    CondWaitMap.remove(obj)

function **lock**(obj) called by thread tid:
    if not primary:
        CurActionList(tid).append(LOCK(obj))
        tid.suspend()  *//until primary*
    if LockedMap(obj) == [tid, n]:   *// reentrant lock*
        LockedMap(obj) := [tid, n+1]; return
    else if LockedMap(obj) == nil:
        LockedMap(obj) := [tid, 1]        // grant lock
    else if LockedMap(obj) == [tid‘, n] and tid≠tid‘:
        MutexWaitMap(obj).append(tid, 1)
        schedule(); tid.suspend()

function **wait**(obj, timeout) called by thread tid:
    [locktid, count] := LockedMap(obj); assert locktid == tid
    if not primary:
        CurActionList(tid).append(WAIT(obj,timeout))
        tid.suspend()  // until primary
    uuid := new unique ID
    LockedMap.remove(obj)    *// fully release lock*
    CondWaitMap(obj).append([tid, uuid])
    schedule()
    if timeout > 0: Timer.setup(timeout, TIMEOUT(obj,tid,uuid))
    tid.suspend()  // until resumed by schedule
    LockedMap(obj) := [tid, count]  *// restore reentrance count*

function **unlock**(obj) called by thread tid:
    if(primary) primaryUnlock(obj, tid)
    else CurActionList(tid).append(UNLOCK(obj))

function **notify[All]**(obj) called by thread tid:
    if(primary) primaryNotify[All](obj)
    else CurActionList(tid).append(NOTIFY[_ALL](obj))

function **Timer.setup**(timeout, message):
    Schedule sending message via abcast after timeout ms
function **Timer.cancel**(message):
    Cancel sending message if not yet sent
```

a reference to the thread to resume and an action list containing the `Timeout` messages is added to `PrimCandidates`.

Lock Operations

The handling of intercepted synchronisation operations is shown in lines 69–100. For `lock()` operations, a thread that is not primary has to suspend until it becomes primary (the suspension is recorded in the action list of the `PrimCandidates` entry that will make the thread primary). As soon as the current thread is primary, it tries to acquire the lock. If `lock()` is called for an already acquired mutex, only the reentrance count is increased (line 74). If the mutex is free, the lock is granted by putting the thread into `LockedMap` (line 76). If it is locked by another thread, the primary thread creates an entry in `MutexWaitMap`, calls `schedule()` and suspends (lines 77–79).

Wait Operations

A `wait()` operation suspends any secondary thread until it becomes primary (lines 83–85). Next, the thread is put into `CondWaitMap`, calls `schedule()`, and suspends. If a timeout for `wait()` is given, the emission of a `Timeout` message is scheduled after the given time (lines 81–91).

Unlock and Notification Operations

The `unlock()`, `notify()`, and `notifyAll()` methods do not suspend a secondary thread. Instead, these operations are simply recorded in the action list of the corresponding `PrimCandidates` queue, and later executed as soon as the action list is processed by `schedule()`. If a primary thread calls these methods, they are executed immediately. Unlock operations decrease the lock counter, and, if the counter reaches zero, remove the thread from `LockedMap` (lines 53–56). The notify operations (`notify()` and `notifyAll()`) move the first element or all elements, respectively, from `CondWaitMap` to `MutexWaitMap`, as the notified threads have to re-acquire the lock prior to continuation (lines 58–67).

Nested Invocations

If the primary thread issues a nested invocation, it calls `schedule()` to select a new primary thread. If a secondary thread issues a nested invocation, this `schedule()` call is delayed until it becomes primary. For this purpose, an action list entry is created. If `schedule()` selects the thread as new primary, it processes the action list and re-calls `schedule()`, as the current thread, which waits for a nested invocation reply, is not available as primary thread. This means that a thread that waits for a nested invocation is never `ActivePrimary`, and it is neither in `MutexWaitMap` nor in `CondWaitMap`. As long as it is waiting for a nested invocation reply, it is an NPC. The arrival of the reply creates a `PrimCandidates` member with a reference to the thread, and makes the corresponding thread a primary candidate (PC).

Termination

When the method finally terminates, this fact is noted in the action list of the `PrimCandidates` queue entry of the thread (line 45).

5.5.4 Verification of the ADETS-MAT Algorithm

Basic Assumptions

For verifying the correctness of the ADETS-MAT algorithm, we assume that all replicas have an identical initial state, no thread is initially active within the replicas and all synchronisation data structures are initialised with an empty state; the sequence of incoming messages is identical in all replicas, and the replica behaviour is piecewise deterministic.

Piecewise Determinism

The piecewise determinism of a replica implementation guarantees that, for an execution interval e_i of thread T (see Definition 5.2 in Section 5.2.3), the local state L_T and the mutex-protected part of the shared state $S_{T,i}$ at the start of e_i uniquely defined the local and shared state at the end of e_i. While the local state only depends on message receptions (client request and nested-invocation replies, which are both consistently delivered to all replicas by total-order multicast) and on previous deterministic thread behaviour, the shared state is also influenced by the activity of other threads. The key problem in verifying ADETS-MAT thus is to show that these activities of other threads take place in a consistent order, which implies that ADETS-MAT creates a deterministic schedule for mutexes.

Lemma 5.3: Deterministic Thread-Execution Interval

The following lemma first shows that each thread-execution interval has a deterministic effect on the scheduling data structures. This is specifically important if a thread-execution interval starts while a thread is not a primary; the executing thread can become primary at a nondeterministic point of time, either at any time during the thread-execution interval, or after the thread has suspend.

Lemma 5.3 (Deterministic Thread-Execution Interval) *Given a consistent local and shared state at the start of the execution interval e_i, the execution of e_i has a deterministic effect on the internal data structures of the scheduling algorithm.*

Verification of Lemma 5.3:

Piecewise Determinism Assumption

(1) The piecewise determinism assumption guarantees that the behaviour of the *replica implementation* is deterministic during e_i.

Determinism after Lock/Wait

(2) If e_i starts by obtaining a mutex lock or by resuming from a `wait` operation, e_i will fully be executed by the primary thread. This means that all intercepted operations will call the same ADETS-MAT functions in all replicas, which will make deterministic modifications to the scheduler data structures.

Determinism of new Requests/after Nested Invocations

(3) If e_i is started by a client request or a nested-invocation reply, the scheduler lets a secondary thread execute the corresponding request, and it creates a `PrimCandidates` entry. This entry can be processed by the scheduler at an arbitrary point in time, and thus the secondary thread can become the primary thread. The transition from secondary to primary thread is not coordinated between the replicas. As a consequence, some replicas can execute an intercepted operation as secondary, while others will execute the same intercepted operation as primary. It needs to be shown that both variants have the same final effect.

(3.1) This claim is obviously true for `wait` and `lock`, as a secondary thread issuing these operations simply blocks until it becomes primary.

(3.2) For `unlock`, `notify`, and `notifyAll`, a secondary thread records the operations in the action list and then, after becoming active primary, executes the same steps as it would have made had it already been active primary.

(3.3) On nested invocations and on thread termination, the active primary thread calls `schedule`, while the secondary instead adds a NESTED entry to the action list. After the secondary becomes primary, the NESTED entry causes the invocation of `schedule`, resulting again in a consistent behaviour.

Consideration of Execution Rounds

In the following, we divide the progress within a replica into *rounds* R_i. Each round starts with the removal of an entry from `PrimCandidates` in `schedule` (line 10) and ends with the next invocation of `removeFirst` in the same line. After initialisation, `schedule` is called when the first element is added to `PrimCandidates`. `MutexWaitMap` is initially empty (line 2), and the invocation of `removeFirst` (line 10) returns the first element from `PrimCandidates`, starting the first round R_1. Subsequent rounds are numbered consecutively. A single round can consist of multiple *execution intervals*.

Lemma 5.4: Consistent ADETS-MAT Behaviour

The following lemma shows that during each round, the ADETS-MAT algorithm behaves deterministically.

Lemma 5.4 (Consistent ADETS-MAT Behaviour) *During round R_i, the ADETS-MAT algorithm will make deterministic selections of the active primary thread and will make deterministic modifications to the scheduler data structures, given deterministic behaviour in all preceding rounds $R_k, k < i$.*

Verification of Lemma 5.4:

Identical Incoming Messages

(1) Initially, the head entry m from `PrimCandidates` is removed. By assumption, the sequence of received messages (and, consequently, of `PrimCandidates` entries) is identical in all replicas.

TIMEOUT Messages

(2) After removing m from `PrimCandidates`, the `schedule` function first processes the action list of m. If the action list contains a `TIMEOUT` message, it does not contain any other entries. The effect of such an entry m is to notify the mutex by deterministically moving it from `CondWaitMap` to `MutexWaitMap`, if the mutex is still waiting. After that, `schedule` is called.

Other Messages

(3) Otherwise, the entry m references a real thread and its action list can contain an arbitrary sequence (zero or more elements) of UNLOCK and NOTIFY/NOTIFYALL entries, optionally followed by a TERMINATE, WAIT, LOCK, or NESTED entry. The referenced thread started with a shared state that only depends on previous rounds, which had a deterministic effect by assumption. By Lemma 5.3, the thread will cause a deterministic scheduler behaviour, independent of the time within its current execution interval at which it becomes active primary. At the end of the execution interval, `schedule` is called.

Activation of Suspended Primary

(4) Subsequently, `schedule` iterates over the entries in `MutexWaitMap`, selecting a new active thread or, if no suitable thread is found, terminating the round. The selection of the new active thread is deterministically defined by the content of `MutexWaitMap` and `LockedMap`. By Lemma 5.3, the interactions of this thread with the ADETS-MAT algorithm will result in consistent modifications to the scheduling data structures in all replicas until the thread suspends or terminates, where it calls `schedule` to re-start the procedure of (4).

Start of Next Round

(5) After the round has terminated, it directly follows from steps 1–4 that the same threads have been selected as active primary thread and that the scheduling data structures at the end of the round are deterministically defined.

Given a consistent initial state, Lemma 5.4 implies by induction on i that the scheduler activates the same threads and makes consistent modifications to its data structures for all rounds.

5.6 ADETS-LSA: A Leader-Follower Algorithm

Algorithm

The loose synchronisation algorithm (LSA) of Basile et al. [BWKI02] uses a leader-follower approach to achieve deterministic multithreaded execution in replicas. The algorithm selects a single replica as primary node. This node records the order in which locks are granted to threads as a sequence of (lock, thread) pairs, and periodically broadcasts this data structure to all other replicas. The original paper evaluates the algorithm with a thread pool containing ten threads and suggests—on the basis of experimental evaluation—that in this setting it is best to send out a message after every ten lock operations. All other nodes suspend threads that request a lock until the corresponding broadcast is received.

Failover Strategy

While the basic operation of LSA is very simple, it requires a strategy to handle the failure of the primary node. Basile et al. define such a fail-over algorithm for crash failures as well as for Byzantine failures. Failure handling requires additional communication between replicas to maintain consistent scheduling. This is a significant difference to other algorithms that do not need any additional computation or communication to handle node failures. An exact description of the failover strategies can be found in the original publication [BWKI02] and is not repeated here.

Extending the Synchronisation Mechanisms

The FT*flex* infrastructure provides the ADETS-LSA algorithm, which extends Basile's basic LSA algorithm. Instead of the simple synchronisation model of LSA, which uses only binary mutexes, ADETS-LSA specifies extensions for fully supporting the native Java synchronisation model. Reentrant locks are implemented in a straightforward way by calling the original lock and unlock operations only for the first lock operation and the last unlock operation of a single thread. Nested invocations do not require any dedicated support in the scheduler implementation, as they do not influence the order of mutex assignments. Support for condition variables and time bounds on blocking wait operations deserves further discussion.

Supporting Condition Variables

The support for condition variables must implement `wait()` and `notify()/notifyAll()` operations. A `wait()` operation simply needs to suspend the current thread, which can be done locally at all replicas. What needs to be made consistent is the relative order of `wait()` and `notify()/notifyAll()` operations on the same condition variable. For example, a thread T_1 might call `condvar.wait()`, while a thread T_2 concurrently calls `condvar.notify()`. If the call of thread T_1 happens first, T_1 will be resumed by the call of T_2. If, on the other hand, the notification happens first, it will have no effect on T_1, and T_1 will remain blocked. A deterministic order of such concurrent operations is easily obtained in all replicas by the LSA algorithm. According to the Java language specification, all operations on a condition variable must be protected by the acquisition of the corresponding mutex. The basic LSA algorithm guarantees a deterministic order of these mutex acquisitions. Hence, the order of operations on the condition variable will be deterministic as well.

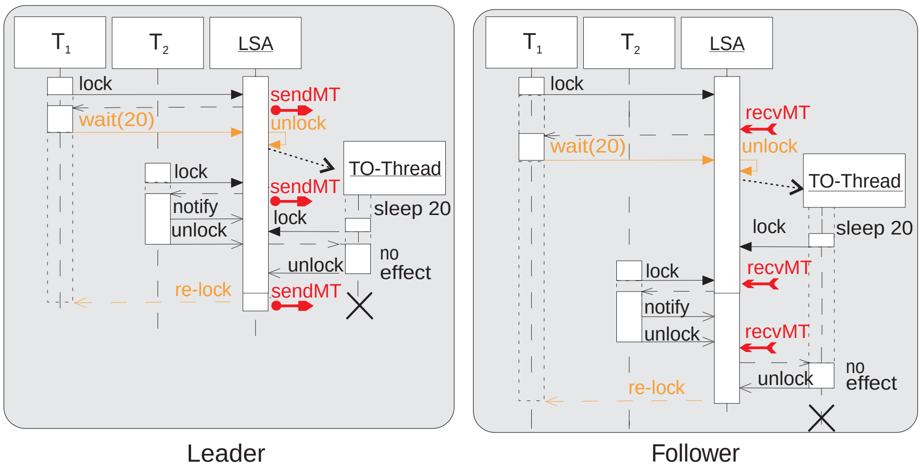


Figure 5.9: Handling timeouts in the ADETS-LSA algorithm

Nondeterminism Caused by Time Bounds

Time bounds on wait operations represent a source of nondeterminism. For example, a thread T_1 might be waiting on a condition variable, having specified a time bound. A second thread, T_2, might call a `notify()` operation. The timeout of T_1 and the notification of T_2 are concurrent; the relative order in which the timeout and the notification happen can result in two possible execution sequences according: either, T_2's notification happens first, which causes T_1 to be resumed by the notification and cancels the timeout; or, the timeout happens first, with the effect that T_1 is resumed by the timeout and, after that, T_2's notification may potentially resume another thread.

No Nondeterminism in ADETS-LSA

Handling such timeouts deterministically requires a non-trivial extension to LSA. In the solution provided by the ADETS-LSA algorithm, a local timeout of a wait operation does not resume the waiting thread directly. Instead, it creates a new thread, which is also subject to the ADETS-LSA scheduling. The thread tries to resume the waiting thread by locking the corresponding mutex and signalling the `wait()` operation to resume.

Sample Execution

A sample execution of this extension is shown in Figure 5.9. Thread T_1 calls `wait()` with a timeout of 20ms. This call causes the LSA scheduler to create a timeout thread (TO-Thread), which sleeps for 20ms and then tries to resume the wait. Concurrently, thread T_2 tries to call `notify()`. Both T_2 and TO-Thread need to lock the same mutex. On the leader node, T_2 is faster, which causes the `notify()` operation of T_2 to resume T_1, and the timeout thread has no effect. On the follower node, the timeout thread requests the lock first. The LSA scheduler, however, records the lock order at the leader node and broadcasts this information to the follower nodes (`sendMT`). The follower nodes grant locks in the order defined by this information (received in `recvMT`), resulting in a deterministic behaviour.

Determinism

The basic scheduling algorithm guarantees that, due to the lock, the signalling is done in a consistent order on all replicas. It is thus deterministic whether the `wait()` is resumed by the timeout or by a different notification. The signalling must be able to check this condition, and therefore the timeout message must carry a unique identifier that specifies the `wait()` operation.

Global IDs For Mutexes and Threads

The original LSA algorithm assumes that globally known IDs for mutexes and for threads exist. Basile et al. describe a method for dynamically adding new mutexes and new threads by explicit notifications sent to the scheduler. For thread creation, this is feasibly in practice: the middleware infrastructure controls the creation of threads and thus can notify the scheduler. There is, however, no explicit creation of mutexes. In Java, every object can be used as a mutex, and there is no globally consistent ID for these objects. Therefore, the ADETS-LSA implementation requires an additional modification to the LSA algorithm.

Consistent Mutex IDs in ADETS-LSA

In ADETS-LSA, the leader replica assigns new mutex IDs automatically on the first lock operation on a not yet known mutex. Follower replicas instead suspend a thread upon a lock operation with an unknown ID. On all replicas, the lock operation can uniquely be identified by the thread ID, as the same thread will lock the corresponding mutex on all replicas. The leader sends its mutex ID with its periodic mutex table broadcast, which enables the follower replicas to learn the new mutex ID.

5.7 ADETS-PDS: Preemptive Deterministic Scheduling

Overview

The preemptive deterministic scheduling algorithms of Basile et al. [BKI03], PDS-1 and PDS-2, are a completely different approach to deterministic multithreading in replicated objects. These algorithms operate in sequential rounds.

PDS-1

In PDS-1, each thread can acquire at most one mutex per round. A thread is suspended when it requests a mutex; as soon as all threads are suspended, a new round is started; as the mutex requests of all threads are known at the beginning of the round, the mutexes can be assigned deterministically to all threads. If multiple threads request the same lock, they get the lock according to increasing thread IDs. For example, if two threads T_1 and T_2 have both requested a mutex m, T_1 may execute and T_2 remains suspended. As soon as T_1 unlocks m, T_2 may execute concurrently with T_1. If T_1 suspends in the current round without unlocking m, T_2 remains suspended.

PDS-2

The PDS-2 variant improves concurrency by allowing threads to acquire up to two locks per round. A round is divided into two phases. Initially, a round starts execution in phase 1 in the same way as PDS-1, granting mutexes according to requests made before the start of the round. If a thread requests a new mutex during phase 1, it is not immediately suspended (as it would be in PDS-1). Instead, this second mutex is granted under the condition that it is available and all threads with lower thread ID have already acquired such a phase-1 mutex. After the mutex acquisition, the thread enter phase 2, in which a mutex requests suspend a thread as in PDS-1. A new round is started as soon as all threads are suspended.

Evaluation

In both PDS algorithms, the number of threads is constant during the execution of a round. New threads may be created or removed only at the start of a new round. Even then, a deterministic rule for changing the set of threads is necessary. The state of the incoming message queue cannot be used for deciding an adjustment of the thread pool size, as the group communication system only ensures a consistent order of message reception, but no consistent time (i.e., some

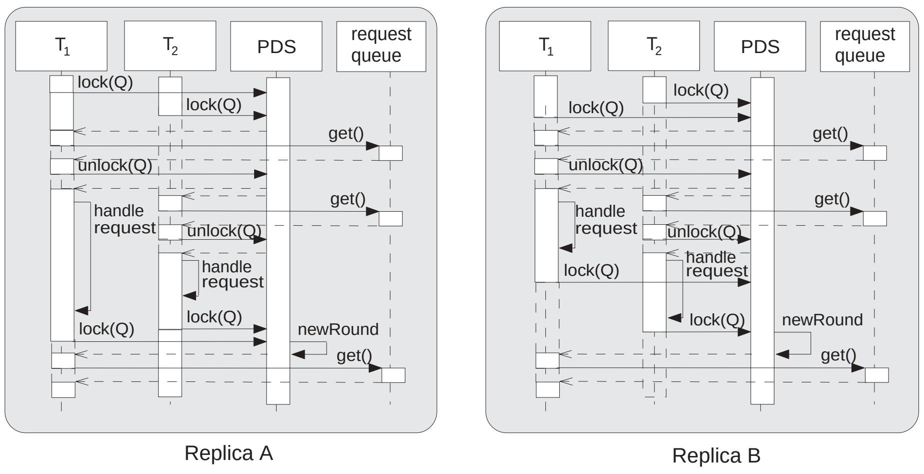


Figure 5.10: Assigning requests to threads in the ADETS-PDS algorithm

replica might already have received a message m that other replicas have not yet received). The PDS algorithms work best if all threads repeatedly execute lock-compute-unlock steps, with each thread having approximately identical computation times. It requires no communication for deterministically assigning mutex locks to threads. The algorithm has two main disadvantages. First, as long as one single thread fails to request a mutex lock, no new round can be started. Second, the number of threads must be known deterministically at the start of each round. Incoming requests have to be mapped to a fixed-size thread pool.

Problem of Request Assignment

The PDS algorithm does not allow an asynchronous creating of new threads for each incoming client request. The original publication simply assumes that sufficiently many requests arrive, so that all threads can continuously execute, without specifying a strategy for assigning requests to threads. In a practical middleware infrastructure, however, such a strategy needs to be implemented.

ADETS-PDS Approach

The ADETS-PDS algorithm uses a *synchronised request assignment strategy*. A thread that has finished processing its last request locks the mutex of the incoming message queue. This mutex lock is granted consistently in all replicas, because this operation is also under the control of the PDS algorithm. Consequently each request is assigned to the same thread in all replicas.

Sample Execution

Figure 5.10 shows a sample execution with two replicas, using the PDS-2 algorithm. In the figure, the threads T_1 and T_2 are executing in the first phase of a round and have just finished processing the preceding request. Thus, they need to obtain a new request. For this purpose, both threads try to acquire the mutex of the message queue. This lock is synchronised by ADETS-PDS, and thus all replicas assign requests to threads identically.

Avoiding Starving Without Requests

If no new requests are available, the system cannot start a new round (as the idling thread will not acquire a lock). The only way to solve this problem is to create artificial requests in case that client requests do not arrive sufficiently frequently.

Extension for Reentrant Locks

Reentrant locks can be added to the basic PDS with the same approach as previously suggested for the LSA algorithm. Only the first lock and last unlock operation of a thread on a mutex is passed to the PDS algorithm, and all additional operations are handled by incrementing and decrementing a counter.

Extensions for Condition Variables

From a consistency point of view, *condition variables* can be supported in the PDS algorithm without much effort. According to the Java language specification [GJSB05], all operations on condition variables must be protected by mutex locks, and thus the relative order of these operations is deterministic on all replicas. A `wait()` operation is added to the PDS algorithms by suspending the thread, in a way similar to a `lock()` operation, combined with an implicit release of the mutex that corresponds to the condition variable. The `notify()` operation must deterministically choose a thread to resume. Such determinism is not guaranteed by the native Java notification mechanisms. Therefore, ADETS-PDS implements its own queue of waiting threads, which is modified deterministically by each `wait()` operation. After the notifying thread releases the mutex associated with the condition variable, the waiting thread can re-acquire that mutex and resume.

Deadlock Avoidance

With a fixed-size thread pool, the use of condition variables can cause deadlocks. If all available threads suspend in `wait()` operations, no more threads are available for handling requests that could resume a waiting thread. To avoid this problem, ADETS-PDS uses a strategy for an automated adjustment of the thread-pool size. The original PDS algorithm supports changing the set of threads at the start of a new round. In a deadlock situation, the conditions for a the start of a new round (i.e., all threads are blocked) are met. Thus, at the start of each round, the number of threads not blocked in a `wait()` operation is compared to a minimum threshold. If the number falls below the threshold, additional threads are added to the thread pool. On the other hand, if there are more non-waiting threads than the threshold and there are insufficient incoming requests (i.e., the request assignment strategy has to suspend a thread temporarily due to the lack of requests), the number of non-waiting threads is reduced to the minimum threshold.

Extensions for Timeouts

ADETS-PDS includes support for `wait()` operations bounded by a timeout. Such timeouts are potentially concurrent with explicit notifications, and thus the algorithm has to make sure that any nondeterminism is avoided. We propose the same concept that we also use for ADETS-SAT and ADETS-MAT. After a timeout occurs, a timeout message is sent to all replicas via group communication. This message is handled by a normal request handling thread, which notifies the waiting thread. As all notifications are synchronised by mutexes, a deterministic order is guaranteed.

Nested Invocations

Nested invocations can be supported in various ways in PDS, but all of them have weaknesses. Most simply, nested invocations can used without specific support by the scheduler. In this case, the thread that waits for a nested invocation blocks all other threads from starting a new round. Alternatively, the scheduler can consider a thread that has issued a nested invocation to be suspended. This enables all other threads to continue executing rounds, but requires a deterministic strategy to resume the thread. For example, if the reply message is processed within some round, the suspended thread can be scheduled for being resumed in the next round.

5.8 Read-Only Operations in the ADETS algorithms

Read-Only Operations

The FT*flex* architecture supports read-only operations that are executed by a single replica only. Handling read-only operations needs support by the thread scheduler.

Synchronisation Model

Read-only operations can only be used for serious purposes if they have access to the object state. Such access needs to be protected from concurrent state modifications. This means that read-only methods have to be able to request and release mutex locks. In the ADETS scheduler implementations, read-only operations are not allowed to use condition variables.

Strategy

Read-only requests are handled by separate threads that are not subject to the usual ADETS scheduling. Instead, all mutex operations of a read-only request are forwarded to a specific ADETS-RO instance that interacts with the current scheduling module.

Read-Only Locks

A read-only thread can acquire a lock if this lock is not held by a modifying thread. The effect of a read-only lock is that the lock is still considered available for internal decisions of the ADETS scheduler. This ensures that the ADETS module makes the same scheduling decision on all nodes, independently on whether they execute the read-only method. If the ADETS module assigns a lock, held by a read-only thread, to a modifying thread, this thread is silently suspended until the read-only thread releases the lock.

5.9 Evaluation

Overview

This section gives an evaluation of the multithreading support in the FT*flex* architecture. A set of benchmarks capture typical interaction patterns of distributed applications. Each benchmark is executed with a pure single-threaded execution model and with all four multithreaded ADETS variants. An object is replicated on three nodes; a constant number of nodes are used in all benchmark, as this number only influences the cost of group communication, but not the behaviour of the scheduling algorithms themselves. Client requests are distributed using the JGroups group communication system [Ban98], as the AGC system has not yet been fully integrated into FT*flex* at the time of measurement. The choice of the group communication system, however, has no significant impact on the relative performance of the scheduling modules, as long as the same group communication system is used for the evaluation of all ADETS variants.

Evaluation Environment

The measurements presented in this section have been made on a set of PCs with a AMD Athlon 2.0 GHz CPU and 1 GB RAM. The PCs have been using Linux kernel 2.6.17 and have been connected by a 100 MBit/s switched Ethernet network. The current prototype of the Aspectix middleware was used on the basis of Sun's Java runtime environment version 1.5.0_03 and JGroups 2.2.9.1.

Benchmark Scenarios

The benchmarks cover four different kinds of scenarios. The first evaluation uses only local computations with lock-protected access to shared state. Next, the scheduler behaviour is analysed in combination with nested invocations. After that, the evaluation addresses the support for condition variables. Last, a complex scenario is considered. A final discussion analyses the advantages and disadvantages of all variants, and provides a basis for selecting an appropriate scheduling strategy for a given application. As there is no single variant that

(a) compute
(b) compute − lock − state access − unlock
(c) lock − state access and compute − unlock
(d) lock − state access − unlock − compute

Figure 5.11: Variants of the local computations benchmark

is clearly superior to all others, it is an important advantage that FT*flex* offers configurability of the ADETS module. In the diagrams, the terms SAT, PDS, LSA, and MAT are used for the ADETS-SAT, ADETS-PDS, ADETS-LSA, and ADETS-MAT module, respectively.

5.9.1 Local Computations

Benchmark Overview

The first group of benchmarks assumes that the behaviour of object methods is limited to performing local computations and requesting and releasing mutex locks. In such a scenario, the only problem of a single-threaded execution is the lack of parallel execution, which primarily is a disadvantage on multi-CPU machines. In the benchmarks, a variable number of clients invoke object methods that have one of the behaviours shown in Figure 5.11.

Description of the Patterns

The pattern (a) does not access the shared object state and thus does not need any mutex access. The pattern (b) first computes and then locks a mutex, updates the object state, and unlocks the mutex again. This is a typical pattern for applications that first perform computations on the request arguments such as verifying digital signatures and preprocessing the client data, and then use this data to update the object state, using a mutex lock to synchronise the update. Pattern (c) is typical for applications that require simultaneous access to client arguments and object state for performing some calculations. The whole request execution is protected by a mutex lock. Pattern (d) can be found in practice mainly for methods that read the shared state and then perform computations (e.g., transformations of state data) to produce the return value for the client.

Simulation Parameters

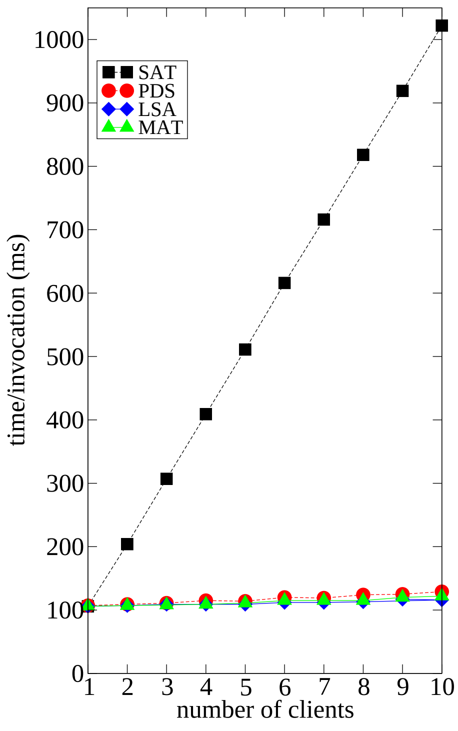
For the following measurements, it is assumed that the local computations take 100 ms. The availability of multiple CPUs is approximated on the single-CPU hardware used for the benchmarks by suspending the request-handler thread for the duration of the computation time instead of performing real computations. It is assumed that the number of CPUs exceeds or equals the number of concurrently computing threads. Furthermore, it is assumed that the methods of the replicated object use fine-grained locking. If all methods used the same mutex lock, this would result in a sequential serialisation. Instead, the benchmarks assume that 10 different mutexes are available, with each client invocation using a randomly selected mutex. The state access itself is assumed to take a negligible amount of time.

Results

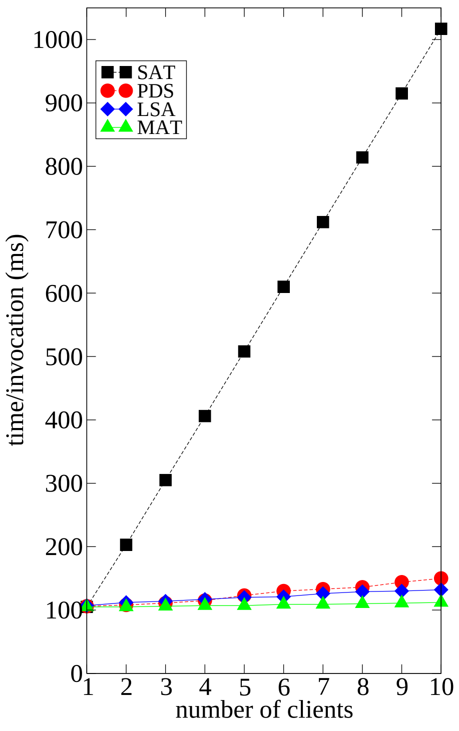
Figure 5.12 shows the result of the benchmarks executed with the remote object replicated on three nodes and accessed by a variable number of clients.

Compute Only

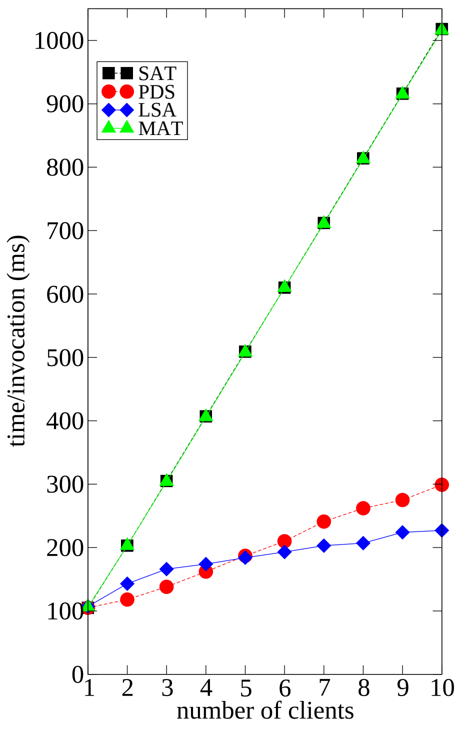
In pattern (a), SAT executes all requests sequentially, while all other variants allow a fully concurrent execution. MAT and LSA perform best, as they can execute all requests immediately in the absence of any synchronisation. PDS shows a small but negligible overhead, because it requires internal synchronisa-



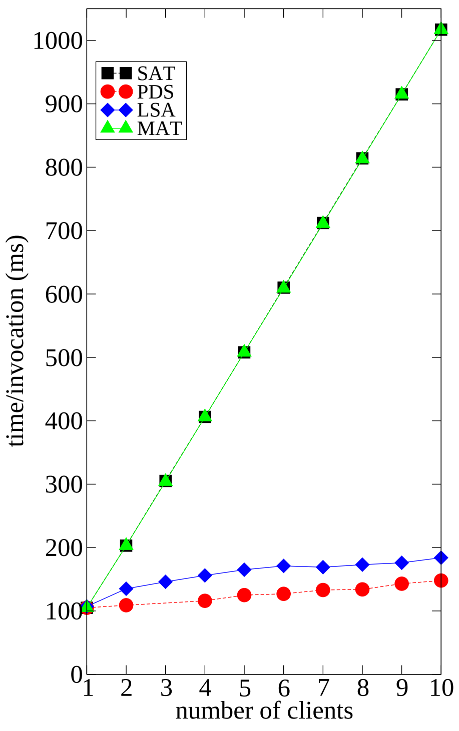
(a) compute



(b) compute–lock–state access–unlock



(c) lock–compute and state access–unlock



(d) lock–state access–unlock–compute

Figure 5.12: Measurements with local computations and mutex locks

tion for assigning requests to threads (i.e., mutex locks for coordinating access to the incoming message queue, see Section 5.7).

Compute–Lock–State Access–Unlock

Pattern (b) results in a similar behaviour. While SAT processes all requests sequentially, all other variants enable a concurrent execution of the computations. MAT is the superior variant, as LSA requires communication for the mutex locks, while PDS uses additional mutex locks for assigning requests to threads.

Lock–Compute and State Access–Unlock

Pattern (c) produces different results. As all requests start with a lock operation and do not define internal scheduling points, the MAT algorithm delays all requests until they become primary. As a result, it serialises all requests, which leads to the same poor performance as the SAT algorithm. LSA and PDS both enable concurrency and show similar behaviour. With an increasing number of clients, the probability that two requests require the same mutex increases. Such a collision causes a sequential execution of both requests within the same internal round and thus delays the start of a new round of the PDS algorithm; thus, with many clients, the LSA algorithm is superior.

Lock–State Access–Unlock–Compute

Pattern (d) is similar to (c); the only difference is that mutex locks are released before the computation. The PDS algorithm benefits from this behaviour, as a collision between two request delays a new round only for the short duration of the state access, and not for the duration of the computation. As a result, PDS is the most efficient algorithm for this pattern, while LSA is slightly slower due to the communication overhead, and both SAT and MAT achieve no concurrent execution.

Comparison of Results

The different benchmark patterns demonstrate that for each algorithm, there are situation in which it performs well, and others in which it does not. Most important, the MAT algorithm is the most efficient one in the situations (a) and (b), while it fails to provide any advantage compared to SAT in the situations (c) and (d). The latter two situations represent worst-case scenarios for MAT. The poor performance of MAT could be alleviated by the introduction of `yield()` operations, which enables a deterministic selection of a new primary thread without reaching an implicit scheduling point. The implemented prototype does not yet provide such an extension.

5.9.2 Nested Invocations

Benchmark Overview

The second set of benchmarks adds nested invocations to the patterns. As explained in Section 3.2, nested invocations can result in deadlocks and reduce performance by causing idle time in a single-threaded execution model. Hence, application patterns with nested invocations are an important scenario even on single-CPU machines.

Simple Benchmark

In the first scenario, two replica groups A and B are created with each consisting of 3 replicas. A varying number of clients call a method at group A, which in turn calls a method at group B. Internally, both requests and the reply from group B to group A are delivered via group communication. This first experiment only uses nested invocation, but no mutex locks or local computations.

Limitation to Sequential and ADETS-SAT

This experiment compares strictly sequential execution with the performance of the ADETS-SAT algorithm. No other algorithms have been evaluated in this benchmark. As there are not lock operations, ADETS-MAT and ADETS-LSA

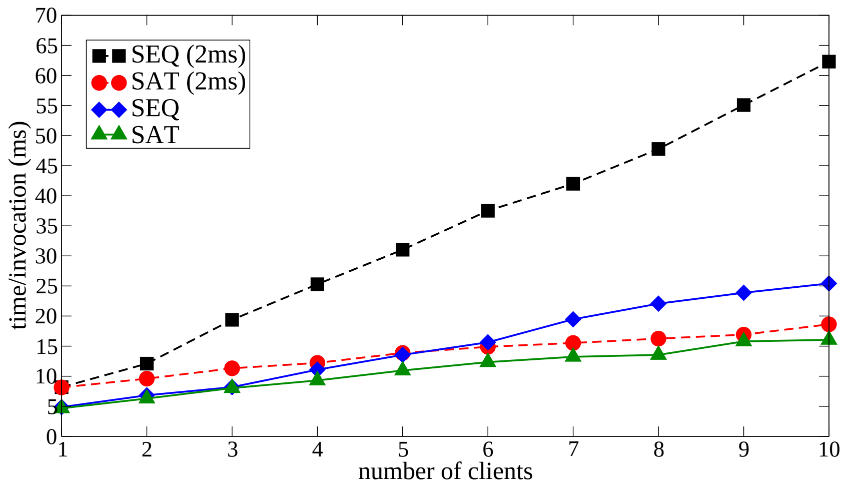


Figure 5.13: Measurements with nested invocations only

would result in a performance similar to ADETS-SAT. On the other hand, ADETS-PDS would require synchronisation for assigning requests to its internal thread pool, adding some overhead.

Results of Simple Benchmark

Figure 5.13 shows the average invocation time measured by the clients, using (a) a strictly sequential execution and (b) the ADETS-SAT algorithm. The solid lines (diamond and triangle symbols) refer to a measurement in which the nested invocation returns immediately. Even in this situation, multithreading with ADETS-SAT is increasingly better with a rising number of clients. In a second measurement (dashed lines with circles and squares), the method called at B suspends for 2 ms before it returns. In this case, the benefit from our multithread approach (which allows accepting new requests at A while the invocation to B is in progress) is even more evident.

Complex Variants

In a second scenario, a set of more complex benchmarks offer a comparison of all ADETS scheduling algorithms. In each benchmark, the replicas execute the following operations:

- nested invocation (duration 100. . .150ms, denoted as **N**)
- local computations (duration 75. . .125ms, denoted as **C**)
- synchronised state updates (lock und unlock operation, denoted as **S**)

The duration of the nested invocations and the local computations was varied over the given interval with uniform distribution.

Measurements

The three elements can be combined in six permutations (NCS, NSC, CNS, CSN, SNC, SCN). Figure 5.14 shows the results of the benchmarks with above parameters, each run with 10 clients.

ADETS-SAT

The ADETS-SAT algorithm performs better than the single-threaded execution, because the idle time of a nested invocation is utilised. ADETS-SAT cannot perform local computations in parallel though. Thus, it performs worse than the other multithreading algorithms.

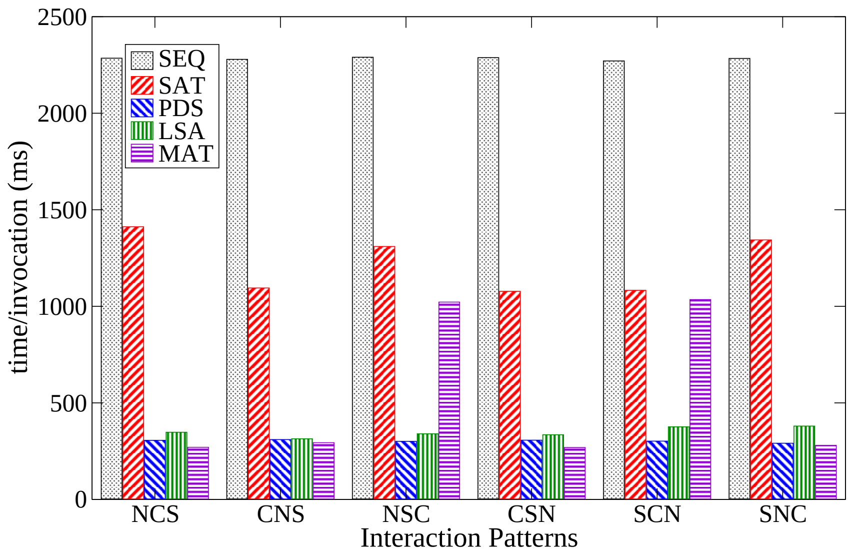


Figure 5.14: Measurements with nested invocations, local computations and mutex locks

ADETS-MAT

The performance of the ADETS-MAT algorithm heavily depends on the interaction patterns. In some situations (NCS, CSN), the algorithm performs best of all. In others (NSC, SCN) it offers no significant advantage compared to the SAT algorithm. The problematic pattern is a state update (S) followed by a computation (C).

ADETS-PDS

The ADETS-PDS performs well in all interaction patterns. The interaction patters do not include the situation that a lock is held during a local computation; as shown in the previous set, this would reduce the efficiency of ADETS-PDS.

ADETS-LSA

ADETS-LSA also performs well in all situations. There is again a slight disadvantage compared to the best algorithm in each benchmark, which can be explained by the communication overhead. This would be worse with increasing network delays.

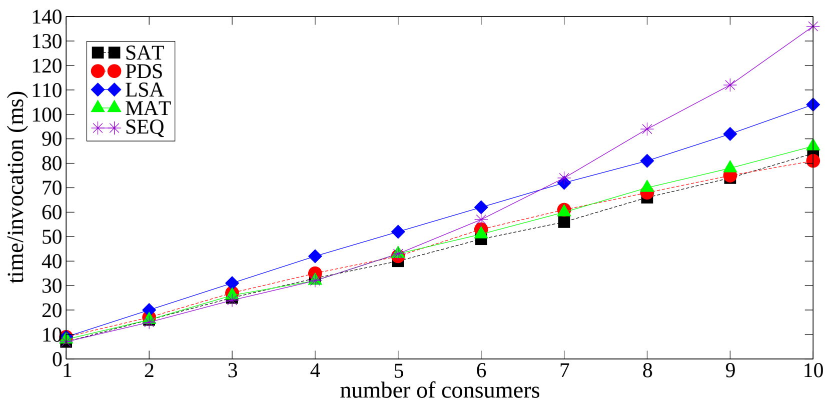
5.9.3 Condition Variables

Overview

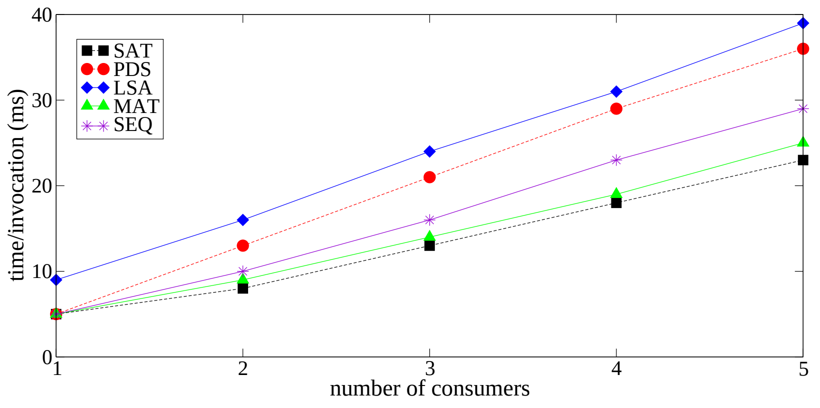
Condition variables are an important mechanism that enables a request to wait for another request. Two different sample applications, an unbounded buffer and a bounded buffer, examine the performance of the scheduling algorithms in combination with condition variables.

Unbounded Buffer

A replicated object that implemented the *unbounded buffer* provides two methods, `consume()` and `produce()`. The `consume()` method returns an available datum, or blocks on a condition variable if no datum is available. The `produce()` method makes a datum available, and notifies another request-handling thread that waits on the condition variable, if such a thread exists. Without support for condition variables, the `consume()` method needs to be implemented differently; to evaluate a sequential execution (which does not permit the use of



(a) unbounded buffer



(b) bounded buffer

Figure 5.15: Measurements with condition variables

condition variables), an alternative implementation of `consume()` with periodic polling is used.

Experimental Results for Unbounded Buffer

Figure 5.15(a) shows the result of this experiment, in which a single producer client and 1–10 consumer clients have been used. With an increasing number of consumers, the single-threaded execution shows an increasing disadvantage. This behaviour is to be expected due to the periodic polling: the number of unsuccessful iterations of `consume()` calls increases with an increasing number of consumers competing for the producer. The other strategies, however, scale linearly because a thread is only notified if a datum in the buffer exists. The ADETS-SAT performs minimally better than ADETS-MAT and ADETS-PDS. The ADETS-LSA, however, has a notable overhead due to the leader-follower communication.

Bounded Buffer

The second benchmark for evaluating the scheduler behaviour in combination with condition variables implements a *bounded buffer*. In this experiment, both `produce()` and `consume()` block if the buffer is full and empty, respectively. Two condition variables are used: the first one is used to resume a blocked `produce()` call by a `consume()` call; the second one is used in the reverse direction.

Experimental Results for Bound Buffer

Figure 5.15(b) shows the result of the experiment, in which the same number of producers and consumers, each ranging from 1–5, have been used. The size of the buffer was set to 2. The graph shows the average time per consumer invocation; exactly the same average time was obtained for producer invocations.

Comparison

Both experiments show that ADETS-SAT and ADETS-MAT are superior to all other execution strategies. ADETS-PDS and ADETS-LSA, on the other hand, show poor performance. In the experiment with the bounded buffer, they perform even worse than the sequential polling-based approach. With the ADETS-SAT algorithm, this is due to the additional communication caused by the scheduling algorithm. With ADETS-PDS, threads that resume from a wait operation need to be delayed until the next internal round starts; this delay increases the invocation times.

5.9.4 Complex Application Patterns

Benchmark with Complex Application Pattern

In the next benchmark, an artificial, complex scenario was tested. The replicated object implements a method that executes 10 iterations of a loop. Each iteration performs the following operations:

- with probability 0.2, a nested invocation (duration approx. 12 ms)
- with probability 0.2, a local computation (duration 10. . .20 ms)
- always, a lock–state update–unlock sequence with a mutex randomly chosen out of a set of 100 mutexes

Fine-Grained Locking

The random selection of the nested invocation and the local computation was used to simulate different kinds of object behaviour. The experiment assumes that the object uses fine-grained locking of the object state; for this purpose, each iteration locks a mutex that is randomly chosen out of 100 available mutexes. In this scenario, all random selections have been determined by the clients to obtain deterministic behaviour in all replicas.

Benchmark Results

The result of this benchmark is shown in Figure 5.16. The single-threaded execution shows the worst performance. The ADETS-PDS strategy also shows poor performance in this scenario. It performs even worse than ADETS-SAT, in spite of ADETS-PDS supporting true multithreading and being able to lock different mutexes concurrently in different threads. The main reason for this is that with ADETS-PDS, if only a single thread performs a local computation, all threads have to wait for this computation to start a new round. ADETS-MAT performs better than ADETS-PDS and ADETS-SAT. The ADETS-LSA algorithm, however, scales best in this scenario, because different mutexes can be locked concurrently by different threads.

5.9.5 Conclusion

Multithreading Superior to Single-Threaded Execution

The benchmarks performed for the ADETS model for multithreaded method execution clearly show that all ADETS variant are superior in performance than a strictly sequential execution.

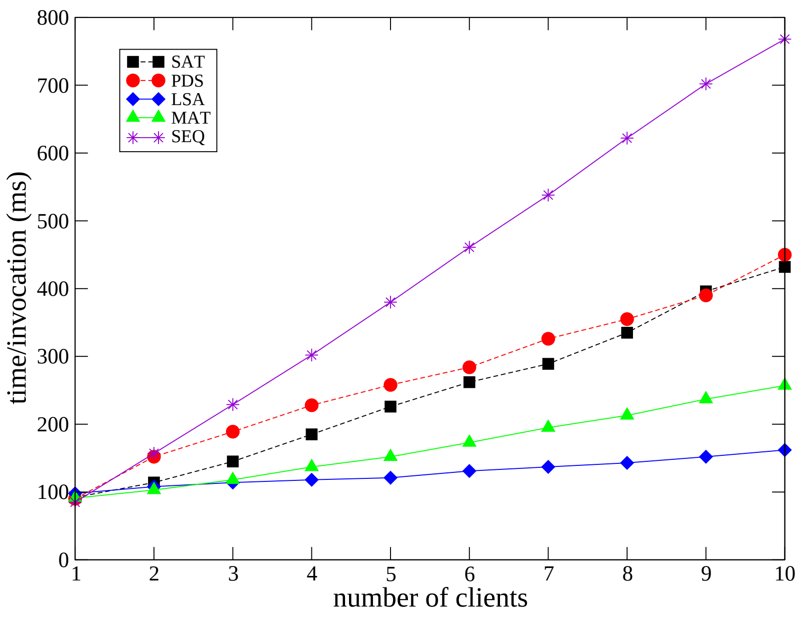


Figure 5.16: Measurements with a complex artificial scenario

No "Best" Variant

Besides that observation, it is evident that there is no "best" scheduling variant. All algorithms are superior in some situations, while providing sub-optimal performance in other situations.

LSA

The LSA algorithm offers optimal concurrency, as it poses no restrictions on the behaviour of the primary node. The primary can execute methods concurrently in the same way as a non-replicated node. It however suffers from its communication overhead. While this overhead can probably be neglected in a high-speed local-area network, it reduces the benefit of multithreading in slower wide-area networks. More severe, the failure of the primary requires (a) accurate and timely failure detection and (b) a costly reconfiguration process; these disadvantages make LSA less useful for active replication of objects.

PDS

The PDS algorithm is a highly specialised algorithm that works fine if replica methods have a behaviour that fits to the period execution in rounds (e.g., each method requests locks a mutex and computes for an approximately constant amount of time). It is not an appropriate algorithm for computations with a highly variable duration, and the use of nested invocations is, in general, less efficient than in the other variants.

MAT

The ADETS-MAT algorithm is the most efficient algorithm in some situation, but performs poorly for requests that request a lock and then compute locally for an extended amount of time without defining a scheduling point.

Future Work

Therefore, an interesting field for future research are improvements of the ADETS-MAT algorithm that maintain its current efficiency, while reducing the problems that the algorithm has in some situations. Such extensions might include strategies that deterministically chose a new primary thread in other situations. Hybrid algorithms that, for example, combine the use of secondary threads—as in ADETS-MAT—with a round-based scheduling–as in ADETS-PDS—might also be interesting.

Benefit from Configurability

One very important contribution of the FT*flex* architecture is the configurability of the scheduling algorithm. This enables the selection of a scheduling algorithm that works best for a given application.

5.10 Summary

Multithreading Support

The FT*flex* architecture supports deterministic multithreaded method execution in replicated objects. It supports a rich synchronisation model, uses code analysis and transformation as a novel way to intercept synchronisation statements, and defines four algorithms for deterministic thread scheduling.

Synchronisation Model

The architecture assumes that access to shared object state is protected by mutex locks. By intercepting synchronisation operations (such as lock and unlock statements), a scheduling module can provide determinism. Unlike other existing systems, the FT*flex* implementation supports reentrant mutex locks, condition variables, and time bounds on blocking wait operations.

Interception by Code Transformation

For intercepting synchronisation operations in replica code, the FT*flex* architecture provides a source-code analysis and transformation tool. In contrast to previous solutions, which intercept synchronisation operations at the operating system level, this approach is independent of the operating system, and is the first system that permits such interception with an unmodified Java virtual machine.

Algorithms

The intercepted operations are delegated to an instance of the Aspectix Deterministic Thread Scheduler (ADETS). Four variants of this module are provided, one for single-active-thread execution on a single-CPU machine, and three variants for true multithreaded execution on multi-core CPUs or multi-CPU machines.

Evaluation

Each ADETS variant has its virtues and problems in specific application scenarios. An experimental evaluation has given a detailed performance comparison of all variants. In all cases, a multithreaded scheduler offers better performance than the sequential request execution that is used in most object replication systems. In addition, the configurability of the ADETS module in FT*flex* offers the possibility to select the optimal scheduler module for a given application.

Chapter 6

Group Communication

Overview The flexibility and reconfigurability of an object replication system highly depends on the flexibility and reconfigurability of the low-level consistency management. The FT*flex* architecture uses totally ordered group communication for active replication. The *Aspectix group communication system* (AGC) provides a reliable, totally ordered multicast mechanism [RBH05]. The main novel features of the AGC are its flexibility in terms of failure model and its support for runtime reconfiguration. The purpose of this chapter is to describe the details of the AGC architecture and to discuss its innovative elements.

6.1 The Aspectix Group Communication System

Consensus-based Group Communication The AGC is a modular group communication system that uses an internal *consensus module* to obtain total message order. Variants of the consensus module support the crash-stop, crash-recovery, and Byzantine failure models. Using a *policy configuration*, the developer can select a specific variant and thus tailor the system specifically for a given application. Furthermore, the system allows consistent reconfigurations at runtime.

Multiple Failure Models The flexible support for multiple failure models is an important advantage of the AGC. Group communication systems such as Ensemble [Hay98], Spread [ADS00], and JGroups [Ban98] assume a fixed crash-stop or crash-recovery failure model. Byzantine fault tolerance is found in intrusion-tolerant group communication systems such as SecureRing [KMMS98] and RamPart [Rei94]. Supporting multiple failure models within a single group communication system not only simplifies the implementation of the middleware infrastructure, but also provides the basis for efficient runtime reconfigurations.

Algorithmic Variants For each failure model, the AGC can support algorithmic variants that differ in their communication patterns and their overhead in case of failures. Thus, the developer can select the optimal variant for a given application, environment, and client interaction pattern. At this level, the AGC provides a configurability that is superior to other existing systems.

Reconfigurability The AGC allows consistent dynamic reconfigurations at runtime. Group members can, for example, decide to replace the instantiation of the consensus module with another one to tolerate a different kind of faults or to adjust parameters that influence performance. These reconfigurations do not cause

```
// Called by external application to start the consensus instance cin
function propose(int cin, value v);

// Provided by extern application, called by the consensus implementation
function decide(int cin, value v);
```

Figure 6.1: Interface of consensus

any service interruption and are transparent to applications. Furthermore, the configuration changes are done in a fault-tolerant way. Hence, node failures cannot cause inconsistent configurations or prevent the reconfiguration.

Self-Optimisation

The ability of changing the system configuration at runtime provides a basis for autonomous self-optimisation. The AGC supports such self-optimisation using an observer/controller pattern. The current prototype provides instrumentation in the internal components of the AGC. Observer/controller modules can access the information obtained from the instrumentation in order to make their reconfiguration decisions. Developers can specify their own implementation of these modules and can thus customise the self-optimising behaviour.

6.2 Consensus-based Total Ordering

Overview

Consensus-based group communication requires a transformation algorithm that provides totally-ordered multicast to the application on the basis of an internal consensus algorithm. The Chandra–Toueg algorithm [CT96] is a well-known approach for such a transformation. While originally intended for closed groups, the algorithm can easily be used with external senders as well. An important contribution of this thesis is the extension of the Chandra–Toueg transformation to systems in which group membership and algorithms can be reconfigured dynamically at runtime.

6.2.1 Consensus Algorithms

Specification of Consensus

Chandra and Toueg [CT96] define the consensus problems in terms of two primitives, *propose(v)* and *decide(v)*. All correct processes propose a value and must reach a common decision. The consensus problem is characterised by the following properties:

- *Termination:* Every correct process eventually decides some value.
- *Integrity:* Every process decides at most once.
- *Agreement:* No two correct processes decide differently.
- *Validity:* If a process decides v, then v was proposed by some process.

Multiple Instances

For implementing a group communication system that continuously enables the transmission of messages, having just a single consensus execution is not sufficient. Multiple, independent instances of consensus are required. A *consensus instance number (cin)* identifies a specific consensus instance. All instances are independent from each other and fulfil the above properties on their own.

```
function init():
    cin := 0
    A_delivered := R_delivered := ∅   // empty list of messages

function abcast(Message msg):   //called by application
    rbcast(msg)

function rdeliver(Message msg):   //called by rbcast
    R_delivered.append(msg)

function processor():                    //called periodically by infrastructure
    if R_delivered \ A_delivered == ∅:
        return
    cin := cin+1
    consensus.propose(cin, R_delivered \ A_delivered)
    suspend(cin)
    A_delivered.append(A_deliver(cin))
    adeliver(A_deliver(cin))

function decide(int cin, MessageSet msgs):      //called by consensus
    A_deliver(cin) := msgs \ A_delivered
    resume(cin)
```

Figure 6.2: Original Chandra–Toueg algorithm

The consensus interface shown in Figure 6.1 includes these consensus instances numbers in the `propose()` and `decide()` operation.

6.2.2 Using Consensus Algorithms for Group Communication

Chandra–Toueg Algorithm

Chandra and Toueg have shown that atomic broadcast and distributed consensus are equivalent problems [CT96]. They propose the algorithm shown in Figure 6.2 to implement atomic broadcast given an implementation of consensus. This broadcast algorithm successively executes independent consensus instances with increasing cin. The k^{th} consensus execution is used to determine the k^{th} batch of messages to be delivered. All consensus messages belonging to the k^{th} instance are tagged with the cin k.

Algorithm Overview

The atomic multicast algorithm internally uses a reliable multicast (*rbcast*). This reliable multicast ensures that a message m is delivered to all group members, but does not guarantee any message order. The interface of *rbcast* is such that the Chandra–Toueg algorithm calls a `rbcast()` method to broadcast a message to all group members, and the reliable, unordered multicast implementation calls `rdeliver()` to deliver the message.

Periodic `processor()` Function

The `processor()` function periodically checks whether there exist messages that the rbcast has delivered via `rdeliver()`, but which the consensus has not decided yet via a `decide()` call; such messages are contained in `R_delivered`, but not in `A_delivered`. If such a set of messages exists, the algorithm advances to the next CIN and proposes this set to consensus. After proposing, the `processor()` thread suspends.

Consensus Decisions

As soon as the consensus algorithm decides the consensus instance *cin*, it calls `decide()`. This function passes the message set from the proposal to the `processor()` thread via the `A_deliver(cin)` variable. The suspended `proces-`

`sor()` thread continues, adds the decided set of messages to the `A_delivered` data structure, and delivers the messages to the application by calling `rdeliver()`.

6.2.3 External Senders

Open and Closed Groups

The Chandra–Toueg algorithm assumes a closed-group model, but it is easily extended for open groups. In a closed group, all actors are group members; in an open group, it is possible that the sender of a message is not a group member. The difference between both variants is only relevant in the `abcast()` function. This function is the only one that the application calls directly for sending a message, and thus external nodes need an implementation of `abcast()`. Internally, the only task of this function is to perform a reliable (but unordered) multicast of the application message to all group members. All other functions of the algorithm are executed only by group members.

Gateway Approach

External senders can perform the *rbcast* using a *gateway approach*. In this approach, the external sender sends its message to a *gateway*. In the AGC, all group members can act as a gateway. The gateway then *rbcast*s the message to all group members in the same way as a group member would send a new message. If the gateway fails during message transmission, the external sender has to contact a new gateway and repeat the message transmission.

Direct Approach

Alternatively, the external senders can perform the *rbcast* using a *direct approach*. In this variant, the sender has to know all group members. This approach is most efficient if the *rbcast* can use available hardware multicast facilities. Without such facilities, the external sender can *rbcast* its message to all group members using direct point-to-point connections. In all cases, the direct approach avoids the delay caused by the gateway approach.

Comparison

Either approach has its virtues in some situations. The gateway approach has the advantage that the external senders need not have accurate knowledge about the current group membership. Instead, they have to know only a single available group member that can act as a gateway. In some scenarios, the gateway approach is very efficient. For example, if the communication group is located in a local-area network, whereas the external senders access the group via a wide-area network, only a single point-to-point communication over the slow wide-area network to the gateway is necessary, and all other communication is done on the local-area network. On the other hand, the direct approach eliminates an indirection step and thus reduces the message latency by one message hop. The AGC prototype implements both variants; the developer can select one of them in the group policy that defines the AGC configuration.

6.2.4 Variability of Membership and Configuration

Necessity

The basic Chandra–Toueg algorithm does not consider the possibility of changing the group membership or the consensus algorithm. The reconfigurability of the FT*flex* architecture, however, requires the possibility of such changes at runtime. Our *ConsensusBcast* algorithm is a novel extension of the Chandra–Toueg algorithm, which enables changes to the internal configuration at runtime.

Group Versions

In the following, the term *group version* is used to refer to a specific *policy configuration*. The configuration specifies the group membership, the consensus algorithm, and additional internal parameters. Each reconfiguration creates a

group version	start cin	policy

0	0	consensus=MultiPaxos
		nodes=$\{P_1, P_2, P_3\}$
		...

1	7	consensus=MultiPaxos
		nodes=$\{P_1, P_2, P_3, P_4\}$
		...

2	15	consensus=BFT_PK
		nodes=$\{P_1, P_2, P_3, P_4\}$
		...

Figure 6.3: Example of a group policy table

new group version; the group version *gv* is a number that is increased by each reconfiguration. The details of this reconfiguration process will be described in Section 6.3.

Associating Consensus Instances with Group Versions

It is necessary to establish an exact relation between consensus instance numbers and group versions. Each consensus instance must have a fixed configuration policy. For this purpose, each group version starts at a specific consensus instances number. A *policy table* stores group version, starting cin, and policy. Figure 6.3 shows a table with three group versions. The AGC maps a consensus instances x to a group version by searching the table for the largest *start cin* that is less then or equal to x. For example, above table maps cin 13 to the group version 1. The *current* group version is defined by the most recent configuration stored in the table.

Extending the Chandra–Toueg Algorithm

The policy configuration specifies the consensus algorithm and the group membership. Therefore, this thesis defines a modified version of the Chandra–Toueg algorithm, which adds information on the group version to all data structures. Our extended algorithm is shown in Figure 6.4, with the extensions being marked in yellow.

Reliable Multicast

The first modification to the Chandra–Toueg algorithm affects the reliable multicast. As soon as a client calls the `abcast()` function, the current group version is assigned to the message. The group version also defines the group membership for the `rbcast()` operation. Upon reception of the reliable multicast in `rdeliver()`, the messages are added to a list of messages specific for this group version.

Normal-Case Operation

Without reconfigurations, the `processor()` function works identical to the original algorithm. In this situation, lines 13–19 have no effect, as there are no messages for old group versions. Furthermore, the current group version is equal to group version of the next consensus instance. Thus, only if there are messages in `R_delivered` but not in `A_delivered` (line 22), these messages are proposed to consensus. The consensus instance to which the value is proposed is dynamically determined via the policy configuration (line 25).

New Versions after Reconfiguration

If a new group version is created, it might happen that the new version is valid starting with a specific consensus instance cin_i, while there are still consensus instances $cin_k < cin_i$ that have not yet been started. In this case, the group version of the consensus instance to be started next is less than the current instance (`cingv < config.currentGV`). It is desirable that the configuration

```
function init():
    cin := 0
    R_delivered := ()    // empty map from group version to messages
    A_delivered := ()    // empty map from group version to messages

function abcast(Message msg):
    rbcast(msg, config.currentGV)

function rdeliver(Version gv, Message msg): //executed by the rbcast protocol
    R_delivered(gv).append(msg)

function processor():    // executed periodically
    lastid := (lowest undecided consensus instance number) - 1
    lastgv := gv(lastid)
    // re-broadcast pending messages from fully decided group versions
    for each oldgv with oldgv < lastgv:
        for each msg in R_delivered(oldgv)  A_delivered(oldgv)):
            R_delivered(oldgv).remove(msg)
            rbcast(msg, config.currentGV))

    cingv := config.getVersionByCIN(cin+1) // group version of next instance
    if cingv >= config.currentGV and R_delivered(cingv)\A_delivered(cingv) == ∅:
        return
    cin := cin + 1
    consensus := config.getConsensusByCIN(cin)
    consensus.propose(cin, R_delivered(cingv) \ A_delivered(cingv))
    suspend(cin)
    A_delivered(cingv).append(A_deliver(cin))
    adeliver(A_deliver(cin))

function decide(CIN cin, MessageSet msgs):    // called by consensus
    cingv := config.getVersionByCIN(cin)
    A_deliver(cin) := msgs \ A_delivered(gv(cin))
    resume(cin)
```

Figure 6.4: The ConsensusBcast algorithm for reconfigurable consensus-based atomic multicast

belonging to the newest group version is used as soon as possible. The original algorithm would propose values for $cin_k < cin_i$ (and thus advance the current cin) only if new application messages are available for being proposed. In the modified algorithm, the `processor` function will instead propose empty message sets for those intermediate consensus instances if no messages are available.

Forwarding Messages to Newest Group Version

After a reconfiguration, there can be messages in `R_delivered(gv)` for some group version `gv` less than the current group version. If all consensus instances of `gv` have been decided, these messages cannot be delivered any more within that group version. Furthermore, these messages cannot simply be used in a newer group version, because the group membership (which is also used in the `rbcast()` operation) might have changed. As a solution, our ConsensusBcast algorithm re-`rbcast()`-s these messages with a new group version.

Multiple Messages per Consensus Decision

Figure 6.5 illustrates the interactions between the system components in the absence of reconfigurations. In the example, the application sends two messages (m_1 and m_2) using `abcast()`, which the ConsensusBcast then distributes via `rbcast()`. Next, m_1 and m_2 are proposed to consensus in a single message set.

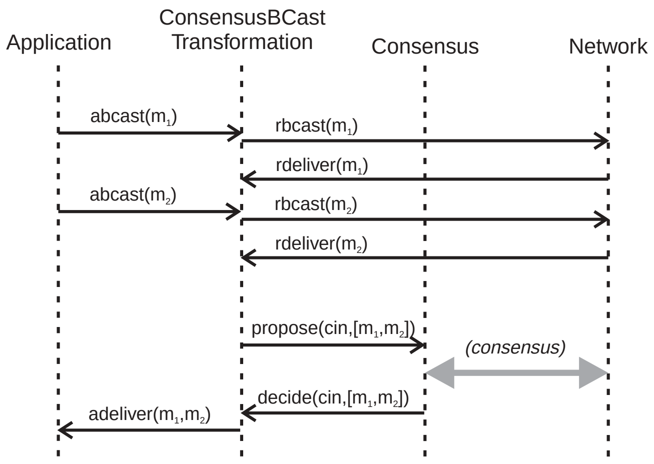


Figure 6.5: Interactions between functional parts of the AGC

This behaviour is configurable: the group policy can define a maximum number of message to be included in a single proposal, and it can define a maximum time to wait between receiving a message in `rdeliver()` and calling `propose()` at consensus with the collected messages. This approach may significantly reduce the overhead caused by the consensus algorithm; however, it increases the message latency by the period of time in which the system waits to collect messages.

Non-Member Membership Updates

External senders that use the direct approach have to assign a group version to their message in the `abcast()` function. The ConsensusBcast algorithm makes sure that if a message is sent by `rbcast()` with some group membership version gv, it is either `adeliver()`-ed with a consensus instance of that gv, or it is re-`rbcast()`-ed with a newer gv. The re-broadcast adds some overhead; optimal efficiency is obtained if the sender of a message uses the current group version. Hence, external members should know about the current group version. Usually, only group members learn about new group version automatically. Therefore, ConsensusBcast implements a simple update strategy for external senders. If a group member receives a *rbcast* from a non-member with an outdated group membership version, it sends an version update message to that client.

6.3 Consistent Reconfiguration

Consistent Reconfigurations

The ConsensusBcast algorithm assumes that there is a mapping from consensus instances to group versions and group policies, with the policies specifying consensus instance, group membership, and internal parameters. The reconfiguration process that creates a new group version must fulfil two constraints:

- A reconfiguration must be made consistently on all group members.
- The configuration of a consensus execution already in progress must not be changed.

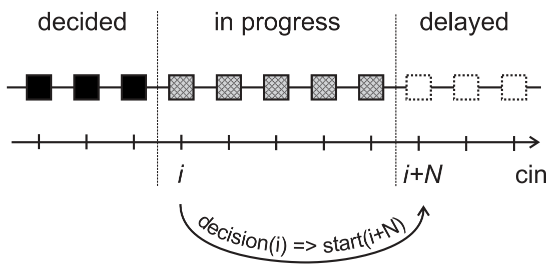


Figure 6.6: Parallel execution of consensus instances

This section addresses the question of how to reconfigure the mapping in the group policy table without violating these constraints.

6.3.1 Consistency on all Group Members

Reconfigurations Sent as Group Message

To perform all reconfigurations consistently on all group members, the AGC sends each reconfiguration to the group as a group message. The semantics of group message delivery ensures that all group members receive the reconfiguration message reliably in an exactly defined position relative to other group messages. Thus, the message not only defines a new configuration policy, but it also determines exactly when the system shall start using the new policy.

Simple Consistency by Sequential Processing

In the simplest variant, the `processor()` function is executed in strictly sequential way. This implies that a consensus instance $i + 1$ is only started after instance i has been decided. In this variant, reconfiguration messages can be applied immediately. The delivery of a reconfiguration by the decision of instance i can, without any problem, change the configuration of instance $i+1$. Thus, reconfigurations can have an immediate effect.

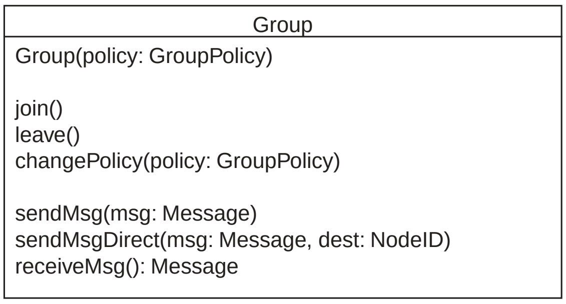
6.3.2 Parallel Execution of Consensus Instances

Parallel Instances

Besides a simple sequential execution, multiple consensus instances can be executed in parallel. This means that a new execution of `processor()` can be started while a previous execution is still in progress. This approach can be used to increase the efficiency, but it complicates the reconfiguration process. It implies that a consensus instance k, with $k > i$, may start executing before instance i has finished its decision. Obviously, inconsistencies will for example arise, if one node starts a consensus instance k with a certain consensus algorithm defined by an old configuration, while another node uses a different algorithm from a newer configuration for the same instances k. Such inconsistencies have to be avoided.

Limitation of Parallelism

More generally, inconsistencies arise if a decision of cin i can modify the configuration of an instance $k > i$ after cin k has started. As a solution to this problem, the ConsensusBcast implementation limits the number of parallel instances and schedules reconfigurations sufficiently far in the future. If a new group policy configuration is decided in consensus instances i, the system activates the new group version in instance $i+N$, N being defined by a group policy. Thus, instances $i \dots i + N - 1$ may all execute in parallel without conflicting



```
Group
Group(policy: GroupPolicy)

join()
leave()
changePolicy(policy: GroupPolicy)

sendMsg(msg: Message)
sendMsgDirect(msg: Message, dest: NodeID)
receiveMsg(): Message
```

Figure 6.7: Application interface of the `Group` component

with the reconfiguration. Consensus instances greater or equal to $i + N$ needs to be delayed until all consensus instances up to cin i have finally been decided. Figure 6.6 illustrates this strategy for $N = 5$. Other group members can already know the decision of cin i and thus might send messages belonging to instance $i + N$; these messages have to be queued until this instance can start locally.

Fast Reconfigurations

Scheduling reconfigurations N rounds into the future introduces a delay. The new configuration is activated only after all intermediate consensus instances are decided. The ConsensusBcast algorithm already contains provisions for this case. As soon as a new group version is determined by a reconfiguration, the transformation algorithm immediately proposes empty message sets for all intermediate decisions for which no proposal has been made yet. Thus, the resulting reconfiguration delay is minimal.

6.3.3 Garbage Collection

Parallel Execution of Old and New Instances

As soon as a reconfiguration starting in *cin t* is decided, instances with old and new configuration (e.g., `ComSys` or `Consensus` instantiation) will operate in parallel. Even if a node has locally decided all instances less than i, it still may not yet discard the old instances, because other nodes might still be executing these instances. A cleanup operation may only be started if all decisions less than i have been finished on all nodes.

Realising Cleanup Operations

To realise such a cleanup operation, a simple garbage collection mechanism is implemented. All nodes periodically send information to the group about the greatest cin, up to which they have finally decided all instances. This information can be sent infrequently using piggybacking on other group messages, which minimises the overhead. As soon as all group members confirm the decision of a cin i, all policies and component instantiations that are not needed by instances greater than i can be cleaned up.

6.4 Prototype Implementation

Implementation Overview

For the application (i.e., for the FT*flex* replication infrastructure), the Aspectix group communication system offers the interface shown in Figure 6.7. This interface offers methods to send messages to the whole group and to individual

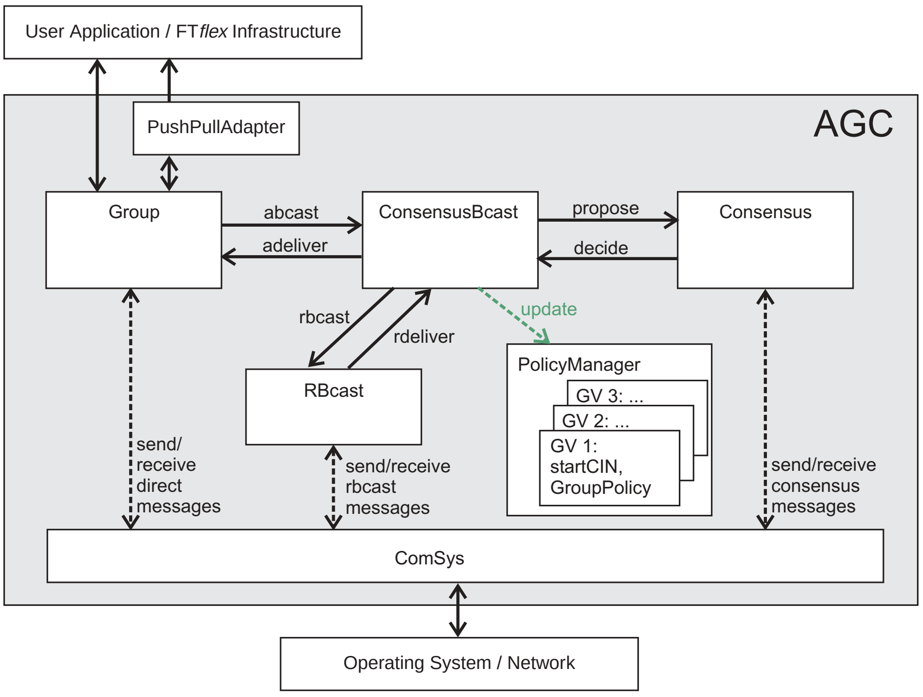

User Application / FTflex Infrastructure
AGC
PushPullAdapter
Group
abcast
adeliver
ConsensusBcast
propose
decide
Consensus
rbcast
rdeliver
update
PolicyManager
GV 3: ...
GV 2: ...
GV 1: startCIN, GroupPolicy
RBcast
send/ receive direct messages
send/receive rbcast messages
send/receive consensus messages
ComSys
Operating System / Network

Figure 6.8: Modular structure of the reconfigurable consensus-based group communication system

members, to receive messages, and to reconfigure the group by adjusting group membership or the group policy.

Group Members and External Senders

Instantiating a new `Group` does not automatically join the group. Instead, the new `Group` instance can be used to access the group as an external sender and does not receive group messages. Calling `join()` turns the external node into a group member, which then starts receiving all group messages. A node can leave the communication group by calling `leave()`; as a result, the node again becomes an external node.

Pull Model

The `Group` interface offers a pull model of interaction at the client interface. This means that clients explicitly have to call a `receiveMsg()` method in order to receive messages. An optional `PushPullAdapter` can be used by the client to receive messages in a push model. In that model, the adapter actively invokes a callback method at the client for each message that is locally received.

Details of the Implementation

The following description explains the internal components of the AGC and the implemented variants of consensus algorithms.

6.4.1 Internal Modular Structure of the AGC

Architecture Overview

The internal modular architecture of the Aspectix group communication system is shown in Figure 6.8. The core component of any communication group is `Group`. `Consensus` implements a consensus algorithm, and ConsensusBcast transforms this algorithm into totally ordered multicast. The `ComSys` component provides low-level communication. The `PolicyManager` component stores group

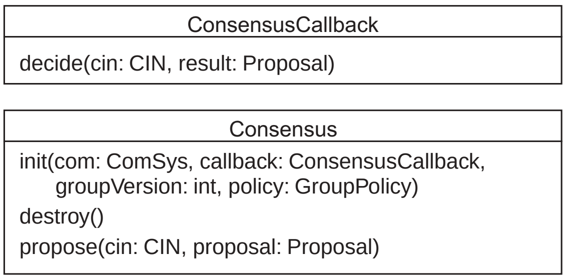

ConsensusCallback
decide(cin: CIN, result: Proposal)
Consensus
init(com: ComSys, callback: ConsensusCallback, groupVersion: int, policy: GroupPolicy)
destroy()
propose(cin: CIN, proposal: Proposal)

Figure 6.9: `ConsensusCallback` and `Consensus` interfaces

versions and corresponding policies, and makes this information available to the other components.

`Group`: Management

The `Group` component implements the application interface. Its main functionality is to pass messages addressed to individual nodes to `ComSys` and to pass group messages and reconfiguration requests to the `ConsensusBcast` module. It also implements the functionality needed for external senders. At an external node, `Group` can forward all client interactions to a group member node. At an internal node, `Group` acts as a gateway that receives the forwarded requests from external senders.

`ConsensusBcast`: Transformation Algorithm

The `ConsensusBcast` component implements the transformation algorithm for realising totally ordered multicast on the basis of consensus algorithms, as has been described in Section 6.2.4. It implements the `ConsensusCallback` interface (see Figure 6.9), which is used by `Consensus` to deliver consensus decisions.

`Consensus`: Consensus Algorithm

The transformation algorithm interacts directly with the `Consensus` component. This component supports executing independent instances of a distributed consensus algorithm, identified by an instance number *cin*, and implements the interface shown in Figure 6.9. The `propose` operation passes a proposal (a collection of group messages or reconfiguration requests) as input to a specific `Consensus` instance, identified by *cin*. A final decision is delivered to `ConsensusBcast` via the `ConsensusCallback` interface. Multiple instances, distinguished by unique *cin* numbers, may execute in parallel. Multiple variants of the `Consensus` component provide support for several failure models. The implemented variants are discussed in the next section.

`RBcast`: Reliable Unordered Multicast

The `RBcast` component implements reliable, unordered multicast. It sends messages from `ConsensusBcast` to all group members, and actively delivers incoming messages from the network via the `rdeliver()` function of `ConsensusBcast`.

`ComSys`: Low-Level Communication

The `ComSys` component encapsulates the specific low-level mechanisms that are used for communication. It provides network-independent addressing, handles message queueing, and re-establishes connections after failures. This component fully supports reconfigurability. Depending on network abilities, different variants like encrypted TLS channels, plain TCP connections, tunnelling via SOAP/HTML, or the use of existing hardware multicast mechanisms can be supported. The group policy defines the instantiation to be used as well as internal parameters, such as timeouts for connection re-establishment. `Con-`

`sensus`, `ConsensusBcast` (via `RBcast`), and `Group` all use the same instance of `ComSys` for low-level communication between the participating nodes.

`PolicyManager`: Configuration

The configuration of all main components (`Group`, `ComSys`, `ConsensusBcast`, and `Consensus`) is described by a `GroupPolicy`. This policy is internally represented as a set of key-value pairs; it defines the group membership, the algorithm used by the consensus instance, the `ComSys` variant, and additional internal parameters. The `PolicyManager` maintains a mapping from group versions to group policies. The details of the reconfiguration process and a sample of the group policy table stored by the policy manager have been given in Section 6.2.4

6.4.2 Consensus Variants

Consensus Algorithms

Many algorithm can be found in the literature that solve the consensus problem. These algorithms are faced with the *FLP impossibility* [FLP85]. The FLP impossibility states that deterministic consensus is impossible in an asynchronous distributed system if at least one node can fail. Practical solutions to the consensus problem can use deterministic algorithms in a synchronous or partially synchronous system model, and they can use randomised algorithms in an asynchronous model.

Current AGC Implementation

The AGC architecture can be used with any distributed consensus algorithm; this allows the use of the most appropriate algorithm in terms of system model and performance characteristics. The current AGC prototype does not use algorithms for a synchronous model, as in most practical scenarios, it is impossible to make strict synchrony assumptions. Randomised algorithm such as the two Ben-Or algorithms [BO83] and the ABBA algorithm [CKS00] work in a completely asynchronous system with crash-stop and with Byzantine faults. These algorithms are more relevant in practice; they are not yet included by the prototype, but could easily be added. The AGC currently implements several deterministic algorithms in a partially synchronous model.

Paxos Variants

More specifically, the current prototype implements several variants of the Paxos algorithm [Lam89], which differ in fault model and interaction pattern. The selection of a variant tailors the system to application requirements and environment properties. The implemented variants are shown in Figure 6.10.

Dummy

The `Dummy` algorithm does not offer a real consensus implementation. Instead, it immediately calls `decide()` for all values that are passed to `propose()`. Using this variant does not enable totally ordered communication. Instead, it only offers reliable unordered multicast (rbcast). This means that using the `Dummy` consensus instance, the AGC can be configured to provide weaker ordering semantics.

Paxos

`MultiPaxosCR` implements Lamport's classic Paxos algorithm. This algorithm assumes a benign crash-recovery fault model and works in three phases. The internal message exchange behaviour is illustrated in Figure 6.11 (a). In phase 1, a leader node merely collects information about values that may have potentially been committed in previous rounds. In phase 2, the leader sends a proposal to the group. This is either the value learned in the first phase, or an external value provided by `propose()`. If sufficiently many nodes acknowledge the reception of the proposal, the leader commits the proposal in phase 3. Usually, the first phase is executed only when a new consensus attempt is started (for example, after a leader change). After the first phase, the algorithm requires three message delays for each consensus decision (Propose, Ack, and Commit).

Class	*Mode*	*Failure Model*	*Comments*
`Dummy`	–		Non-uniform rbcast without delivery guarantees; only used for analysing overhead of transformation algorithm
`MultiPaxos`	standard	crash-stop	Total order with standard Paxos using majority quorums
	fast	crash-stop	Fast variant of Paxos, with higher network load but potentially less latency
`MultiPaxosCR`	standard	crash-recovery	Standard Paxos recording all essential state changes on hard disk, thus enabling recovery after failure.
	fast	crash-recovery	Fast variant of Paxos with state recording and recovery.
`BFT_PK`		Byzantine	Castro's algorithm with public-key cryptography.

Figure 6.10: Implemented consensus variants

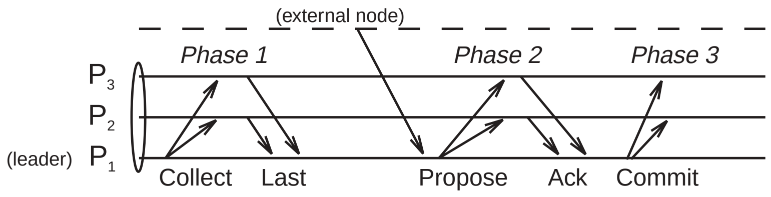
One additional message delay arises from the necessity to send the proposal to the leader node. The algorithm has to write essential state information to stable storage. This information is necessary for consistent recovery of nodes after failures. The current prototype uses synchronous writes to the local hard disk as stable storage.

Fast Variants of Paxos

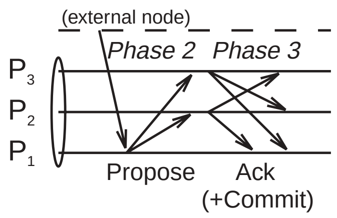
The idea of Brasileiro et al. [BGMR01] can be used to obtain fast variants of the Paxos algorithm, as shown in Figure 6.11 (b) and 6.11 (c) (Phase 1 is omitted in both cases, as it is identical to the standard mode). Only the fast variant (b) has been implemented in the current prototype. In this variant, the Ack and Commit messages are combined by broadcasting the Ack to all group nodes, which in turn may decide autonomously if sufficient Acks are received. This variant reduces latency at the cost of an increased number of messages to be sent. In the ultrafast variant (c), a proposal initially is sent not only to the group leader, but to all nodes, resulting in the elimination of the Propose phase. If sufficiently many nodes send an Ack for the same proposal, they may commit immediately with only one communication step. If not—which may happen if clients propose several values concurrently—a conflict is detected, and the algorithm reverts to classic Paxos. This variant improves latency even further in optimistic cases, at the cost of reduced performance in case of conflicts caused by concurrent access.

Crash-Stop vs. Crash-Recovery

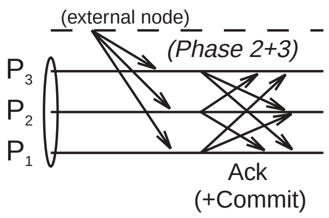
The Paxos algorithm originally was proposed for a crash-stop failure model. Our implementation is based on the modularisation of Paxos, as described by Boichat et al. [BDFG03]. The modularisation offers a simple way to obtain variants. One variant is the elimination of write operation on stable storage, which results in an algorithm for the crash-stop failure model (instead of the crash-recovery model).



(a) Standard mode



(b) Fast mode



(c) Ultrafast mode

Figure 6.11: Speed variants of the generic Paxos implementation

Byzantine Failures

Castro's algorithm [Cas01] extends Paxos for *Byzantine node failures*. This algorithm has an interaction pattern that requires three communication steps in normal-case operation (i.e., in Phase 2 and 3). The current AGC prototype implementation in the `BFT_PK` consensus module supports the Castro algorithm based on public-key authentication. It is furthermore possible to reduce the latency for Byzantine consensus to two communication steps [Zie04]. This variant has not yet been integrated in the AGC prototype. In our experience, the message latency of Castro's algorithm is dominated by the computation time for public-key signatures. Thus, reducing the network delay will have only little practical impact in most situations.

Dimensions of Configurability

The presented variants of Paxos yield two dimensions of configurability: fault model and speed. The fault model is mainly subject to the application's requirements; dynamic reconfiguration is only necessary if the application administrator explicitly requests a change. Different speed variants exist for all fault models. The optimal selection is primarily subject to network properties and application interaction patterns. As these conditions may easily change at runtime, a dynamic reconfiguration is necessary; such a reconfiguration may either be triggered manually, or it can be performed automatically using predefined action rules.

6.4.3 Provisions for Self-Optimisation

Basic Infrastructure

The reconfigurability of the Aspectix group communication system can be used to provide mechanisms for autonomous runtime adaptation. For this purpose, the AGC provides a basic infrastructure that can be extended with custom adaptation strategies. The implementation of specific adaptation strategies are left to the developer and are not discussed by this thesis.

Interfaces

Adaptation strategies need access to internal statistical information that can be used to trigger adaptations. Therefore, the `Group` provides a `getStats` method that returns a set of key/value pairs with internal statistics. The statistics are collected from all internal components (i.e., all internal components also implement such a `getStats` method, which are combined to provide the final result).

Monitoring Data

The content of the monitoring data is individually defined by the components. This makes it possible, e.g., to collect internal data in a Paxos or FastPaxos instance of the `Consensus` module, and provide an observer/controller module that uses these algorithm-specific data to switch between the two variants. In the `Group` and `ComSys` component, information about internal message queue lengths is provided. In addition, such observer/controller module can access other external entities such as the network load or the CPU usage. Such entities are not considered in the current prototype. In ongoing work, which is not part of this thesis, a distributed resource management framework [KHR04] is being developed for the Aspectix middleware, which will contain functionality to obtain such profiling information.

Autonomous Adaptation

It is questionable if autonomous adaptation of the failure model is useful in practice. What type of failures the system should be able to tolerate is a decision that is made by the application administrator. This can be interpreted to mean that autonomous adaptation is not desirable. There are, however, situations in which autonomous adaptation can be advocated for increasing fault tolerance. For example, with four replicas of some service, a replication protocol could enable the toleration of a single Byzantine failure. If only three replicas of this service exist, the protocol is no longer able to tolerate any benign or malicious failure. Switching to another protocol that is able to tolerate the crash of one of the three nodes (and switching back to the Byzantine protocol as soon as four replicas are available) increases the availability of the application.

6.5 Evaluation

Overview

This section evaluates the Aspectix group communication system (AGC). Measurements analyse the performance and the scalability of the current prototype implementation. Furthermore, a discussion compares the properties of the AGC in failure situations with other approaches.

6.5.1 Measurements

Scope of the Measurements

Two aspects are important in a reconfigurable group communication system: the efficiency of sending plain group messages in such a reconfigurable implementation and the cost of reconfiguration operations. Sending normal group messages is the dominant operation in a group, while reconfiguration usually will happen far less frequently. For this reason, the measurements focus on the normal-case efficiency of the reconfigurable architecture in various configurations. All experiments described below have been carried out on Intel Pentium 4 (3.0 GHz) workstations running Linux (kernel 2.4.30), connected via a switched 100-BaseT network, and using TCP channels for low-level communication.

Efficiency of Consensus Implementations

The most important factor in system performance of a consensus-based group communication system is the efficiency of consensus decisions. Figure 6.12 shows

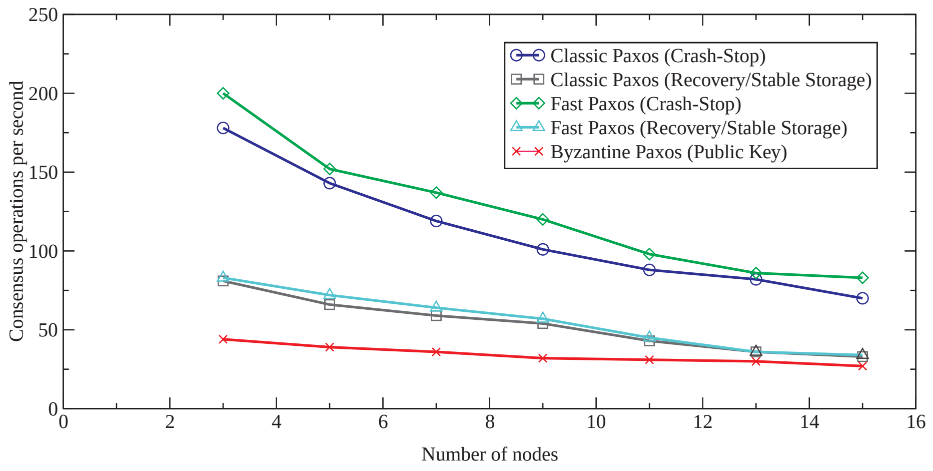


Figure 6.12: Measurements of consensus operations per second

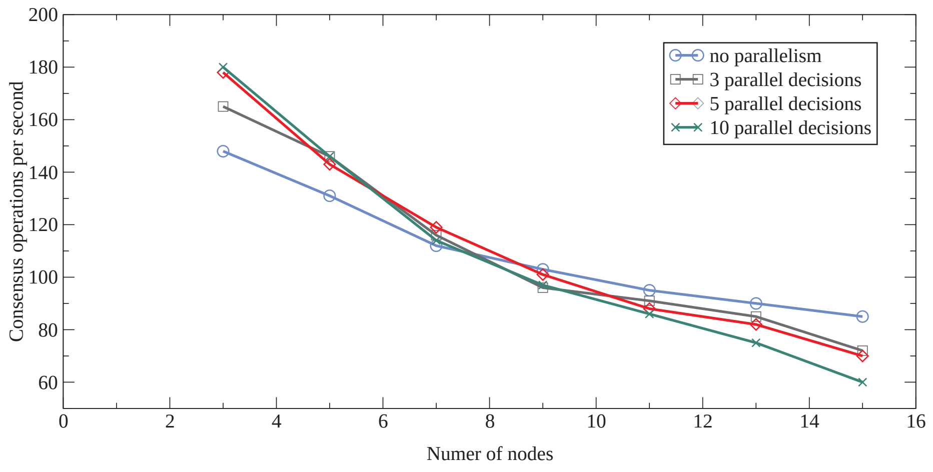


Figure 6.13: Measurements of the effect of parallelism on classic Paxos (crash-stop)

the number of consensus decisions per second that the AGC system achieves in relation to the number of core group nodes, for various consensus instantiations. The recovery variants use synchronous writes to the local disk as stable storage; the parallelism of consensus decision was limited to five parallel instances.

Discussion of Results

For all variants, the system scales well with an increasing number of nodes. The limiting factor in the stable-storage variants is the synchronous write operations; thus there is only little difference between Classic Paxos and Fast Paxos in these cases. The Byzantine consensus instance is, as it might be expected, the most costly variant. The current prototype uses public-key based signatures; it does not yet support the more efficient variant of Castro [Cas01] on the basis of symmetric message authenticators.

Parallel Consensus Decisions

Parallelism of consensus operations, as explained in Section 6.3, makes reconfiguration a slightly more complicated task. Therefore, we examined in another experiment the benefit of such parallelism. Figure 6.13 shows the number of consensus decisions per second for different degrees of parallelism. For a small

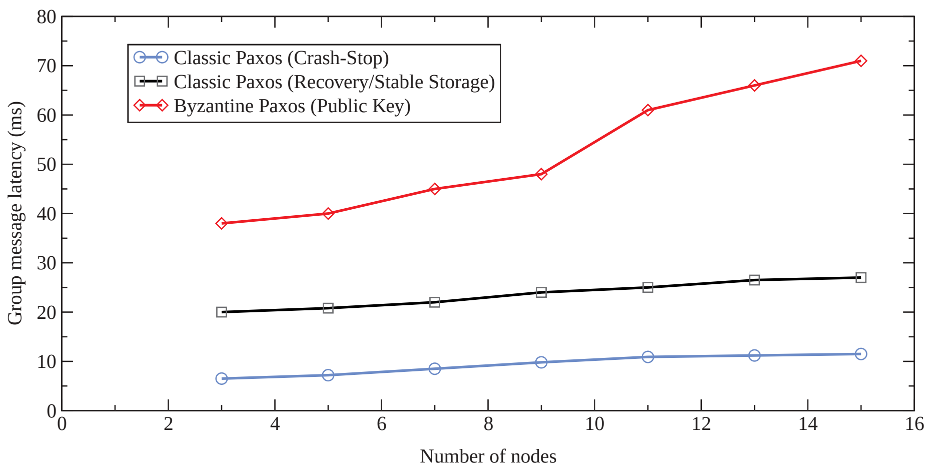


Figure 6.14: Measurements of the group message latency with different consensus algorithms

number of nodes (less than nine), such parallelism increases the performance. Somewhat unexpected, the performance decreases at a higher number of nodes, which is probably due to the overhead of internal synchronization. The results thus also show that a dynamic configuration of the parallelism depending on the number of nodes is necessary to get optimal performance. Hardly any difference exists between five and ten parallel rounds, which coincides with the exception that a small number of parallel rounds is always sufficient.

Message Latency

From the application point of view, an essential parameter is the message latency. Figure 6.14 shows the latency for three different consensus variants, depending on the core group size. All times are per-message latencies averaged over 100 messages sent to group, measured at the application-level group interface.

6.5.2 Discussion of AGC Behaviour in Failure Situations

Comparison with Other Approaches

For relating the AGC system with other group communication system, the characteristics of consensus-based group communication have to be re-iterated. Most existing group-communication systems advocate a sequencer-based approach (see Section 3.3). These systems are simple and efficient; their disadvantage is the single point of failure that the sequencer represents.

Failure Handling in Consensus-based Group Communication

Seamless fail-over mechanisms need an accurate failure detection, which generally is used both for ensuring liveness of the message ordering strategy and for managing the group membership. Consensus-based group communication hides the fail-over complexity within the consensus algorithm. Some consensus algorithms work even fully decentralised. Others use a designated coordinator node internally; yet in the latter case, consensus can use internal timeouts that yield fast but inaccurate failure detection, which can be sufficient to ensure liveness. At the group level, other failure-detection strategies—ranging from conservative timeouts to manual management—can be used for managing group membership.

Advantages of Consensus-based Group Communication

The separation of group membership management and failure detection for obtaining liveness is a key advantage of the AGC. Short timeouts can be used internally to ensure fast message delivery even in failure situations, while no group members can erroneously be excluded from replica groups due to random message delays.

6.6 Summary

Consensus-based Total Order

The Aspectix group communication system (AGC) provides a totally ordered multicast mechanism. It has a modular structure and realises consensus-based total order.

Flexible Failure Models

The AGC supports multiple failure models. The developer can select between a crash-stop, crash-recovery, and Byzantine model. This way, the same basic architecture and the same interfaces can be used for multiple application scenarios. Thus, the FT*flex* infrastructure can easily support various failure models.

Variability

For each failure model, several algorithmic variants can be used, which allows the system to be tailored to the specific application and the network environment. The current prototype implements a "normal" and a "fast" variant of the Paxos algorithm, both in the crash-stop and in the crash-recovery failure model.

Runtime Reconfiguration

On the basis of the given variability, the AGC supports consistent runtime reconfiguration. Ordering protocols are extended with mechanisms that, without violating ordering guarantees or other semantic properties of group communication, allow the replacement of internal components. This feature is also provided in a fault-tolerant way, which means that the system can tolerate node failures during the reconfiguration process.

Self-Optimisation

The given reconfigurability feature can also be used to implement self-optimisation. All AGC components provide simple monitoring interfaces that allow external observer/controller modules to retrieve internal information about message arrival rates, failure frequency, and queue lengths. The observer/controller module can then request a new configuration at the group. This way, extensible basic support for autonomous adaptation is provided.

Chapter 7

Conclusions

Main Problem Areas

This thesis addresses three main problems in the area of fault-tolerant object replication in distributed systems: middleware integration and development support; deterministic multithreaded execution of object methods; and reconfigurable group communication.

Middleware Integration and Development Support

For middleware integration and development support, this thesis presents a solution on the basis of the fragmented-object model. The solution advocates the use of semantic annotations and automated code generation to simplify application development.

Deterministic Multithreading in Replicated Objects

For deterministic multithreaded execution of object methods, this thesis specifies and evaluates a set of deterministic scheduling algorithms. In addition, it proposes code transformation as a new way for intercepting synchronisation statements in the object implementation.

Flexible and Reconfigurable Group Communication

For totally ordered group communication, this thesis presents the design of a consensus-based group communication system, which offers flexible support for various failure models and for dynamic runtime reconfiguration.

7.1 Contributions

Fragmented Objects

The proposed architecture supports active replication for CORBA applications. It uses the fragmented-object model of the Aspectix middleware and provides a customisable replication infrastructure. The architecture enables portable, efficient, and transparent mechanisms. Furthermore, it supports runtime reconfiguration and autonomous adaptation.

Semantic Annotations

The FT*flex* architecture enables the developer to explicitly specify semantic knowledge by annotating objects on a per-method basis. These semantic annotations provide structural and behavioural information. Structural annotations can be used to locate parts of the object functionality directly at the client side. Behavioural annotations specify semantic properties, for example that a method is read-only. These annotations can be used by the infrastructure to optimise the replication mechanisms.

Code Generation

We propose automated code generation as a means to provide custom replication code for each replicated object. The custom code includes a client-side *access fragment* and a server-side *replica fragment*. Unlike other infrastructures, which create code on the basis of mere interface definitions, the FT*flex* code generation tool considers semantic annotations to produce optimised code.

Multithreading

A major contribution of the presented FT*flex* architecture is the support for multithreaded method execution in replicated objects. Multithreading lacks support in most existing object replication infrastructures. Such a support, however, is necessary to avoid potential deadlocks, to increase the performance, and to simplify the reuse of existing servant implementations.

Java Synchronisation Model

The multithreading support of FT*flex* uses a synchronisation model that includes reentrant mutexes, condition variables, and time bounds on blocking wait operations. This enhances the flexibility compared to other middleware platforms with multithreading support, which allow only binary mutexes. The FT*flex* model is comprehensive enough to offer full support for the native Java synchronisation model. Hence, the development of servants and the re-use of existing implementations is simplified.

Interception by Code Analysis

Deterministic multithreading support requires the interception of synchronisation operations of the replicas. For such an interception, this thesis proposes the use of source-code analysis and transformation. Unlike other systems, which rely on interception in the operating system or in thread libraries, our approach does not require low-level system support.

ADETS Scheduler Implementations

The intercepted synchronisation operations are delegated to an instance of the Aspectix Deterministic Thread Scheduling (ADETS) module. This thesis specifies four variants of this module. ADETS-SAT offers a simple, single-active-thread execution model; ADETS-LSA and ADETS-PDS extend the existing LSA and PDS algorithms for our system model; and ADETS-MAT uses a completely new algorithm that, in specific situations, is more efficient than previously used approaches.

Flexible Group Communication

At the level of consistency management, this thesis contributes a consensus-based group communication system (AGC). The AGC has a modular structure that allows the integration of various consensus algorithms. Hence, the system can support multiple failure models, ranging from crash-stop to Byzantine, and algorithmic variants that allow tailoring the system for the application and the environment.

Group-Communication Algorithm

A major contribution of the AGC is an extended algorithm for transforming consensus algorithms into totally ordered group communication. This transformation algorithm enables the dynamic replacement of the consensus algorithm, the adjustment of group membership, and the modification of other group configuration parameters.

Reconfigurability and Adaptivity

The AGC system uses a policy-based mechanism for runtime reconfiguration. The reconfiguration process makes sure that new configurations are activated consistently in all replicas, and that configuration changes are coordinated with internal activities. Furthermore, simple instrumentations provide a basis for autonomous self-adaptation.

7.2 Limitations and Future Work

Object Replication

The FT*flex* architecture addresses the domain of fault-tolerant object replication. Most importantly, the fragmented-object approach, together with the semantic annotations and the code generation, is specific for the object-replication domain. The ADETS scheduler family is also designed for the concurrent execution of methods of replicated objects. The ADETS prototype is used in the FT*flex* infrastructure; the multithreading concept itself, however, could easily be applied to other object-replication platforms. Finally, the AGC system provides generic group communication facilities. As such, it is not limited to object replication and could directly be used for other purposes.

Fragmented-Object Middleware Required

The FT*flex* infrastructure assumes that the basic middleware supports fragmented objects. Popular middleware systems usually do not include such support. We have shown, however, that fragmented-object support can be added to existing object-based middleware systems with little effort. For CORBA-based systems, we have proposed a refactorisation that separates reference handling from the ORB core, enabling the implementation of fragmented-object support in a generic plug-in module [HKRS05]. For platforms such as Java RMI and .NET Remoting, the support for fragmented objects can even be added transparently without internal middleware modifications [KDH$^+$06, RDH05].

Focus on Active Replication

The FT*flex* prototype currently supports only active replication. While this replication style has many benefits, reasons such as constrained computational resources and nondeterministic behaviour can favour other schemes such as passive replication. A production system should include variability in the replication model; such variability could be integrated into the reconfiguration facilities of the current FT*flex* infrastructure.

Formal Verification

This thesis does not provide a complete formal verification of its components. The verification of the general architecture, of the ADETS family, and of the AGC is based on informal correctness arguments and on empirical tests. A better way of verification would be based on a formal verification. For example, the presented algorithms for scheduling and group communication could be specified in a language such as CSP and automatically shown to be correct by verification tools. Such an approach could provide further reasons for confidence in the FT*flex* architecture.

Weaker Semantics

The infrastructure currently provides strict replica consistency. For some applications, weaker semantic models could provide an efficiency gain. It would be interesting to extend the concept of semantic annotations to the client side, thus enabling client developers to specify the semantics that the client demands from the replicated object.

Other Synchronisation Mechanism

Instead of using locks to synchronise concurrent threads in multithreaded execution, mechanisms such as lock-free synchronisation could be considered. Furthermore, approaches to synchronisation that are less nondeterministic than traditional multithreading, such as the use of explicit rendezvous points for state exchange between threads, could be integrated.

List of Abbreviations

ADETS	Aspectix DEterministic Thread Scheduler
ADK	Aspectix Development Kit
AGC	Aspectix Group Communication
API	Application Programming Interface
CAS	Compare And Swap
CIN	Consensus Instance Number
CORBA	Common Object Request Broker Architecture
FIFO	First In, First Out
FO	Fragmented Object
FT-CORBA	Fault-Tolerant CORBA
GIOR	Group-IOR
ID	Identifier
IDL	Interface Definition Language
IOR	Interoperable Object Reference
JVM	Java Virtual Machine
LAN	Local Area Network
LSA	Loose Synchronisation Algorithm
MAT	Multiple Active Threads
NPC	Non-Primary Candidate
OMG	Object Management Group
ORB	Object Request Broker
PC	Primary Candidate
PDS	Preemptive Deterministic Scheduling
PECSAR	Portability, Efficiency, Client Transparency, Servant Transparency, Adaptivity, Reconfigurability
RMI	Remote Method Invocation
ROWA	Read One Write All
SAT	Single Active Thread
SLT	Single Logical Thread
WAN	Wide Area Network

Bibliography

[ACKM03] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju, *Web services - concepts, architectures and applications*, Springer, November 2003.

[AD76] Peter A. Alsberg and John D. Day, *A principle for resilient sharing of distributed resources*, ICSE '76: Proceedings of the 2nd International Conference on Software Engineering (Los Alamitos, CA, USA), IEEE Computer Society Press, 1976, pp. 562–570.

[ADS00] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton, *A low latency, loss tolerant architecture and protocol for wide area group communication.*, DSN, IEEE Computer Society, 2000, pp. 327–336.

[Bal90] Henri Bal, *Programming distributed systems*, Silicon Press/Prentice Hall, 1990.

[Ban98] Bela Ban, *Design and implementation of a reliable group communication toolkit for Java*, Tech. report, Dept. of Computer Science, Cornell University, 1998.

[Bau06] Peter Baumann, *Konzeption und Implementierung eines Systems zur verteilten Quellcode-Verwaltung*, Student Thesis, Department of Distributed Systems and Operating Systems, University of Erlangen-Nürnberg, Germany, 2006.

[BBH$^+$98] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek, *Performance evaluation of the Orca shared-object system*, ACM Transactions on Computer Systems **16** (1998), no. 1, 1–40.

[BCBT96] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg, *Simulating reliable links with unreliable links in the presence of process crashes*, Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96), 1996, pp. 105–122.

[BCH$^+$98] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik, *Filterfresh: Hot replication of Java RMI server objects*, Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS) (Santa Fe, New Mexico), April 1998, pp. 65–78.

[BDFG03] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui, *Deconstructing Paxos*, SIGACT News **34** (2003), no. 1, 47–67.

[BG84] Philip A. Bernstein and Nathan Goodman, *An algorithm for concurrency control and recovery in replicated distributed databases*, ACM Trans. Database Syst. **9** (1984), no. 4, 596–615.

[BGMR01] Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal, *Consensus in one communication step*, Proceedings of Parallel Computing Technologies, 6th International Conference, PaCT 2001, Novosibirsk, Russia, September 3-7, 2001, 2001, pp. 42–50.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley, 1987.

[Bir97] Kenneth P. Birman, *Building secure and reliable network applications*, Manning Publications Company, Greenwich, CT, 1997.

[BJ87] Kenneth P. Birman and Thomas A. Joseph, *Exploiting virtual synchrony in distributed systems*, SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles (New York, NY, USA), ACM Press, 1987, pp. 123–138.

[BKI03] Claudio Basile, Zbigniew Kalbarczyk, and Ravi Iyer, *A preemptive deterministic scheduling algorithm for multithreaded replicas*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2003., 2003, pp. 149–158.

[BMS02] Martin Bertier, Olivier Marin, and Pierre Sens, *Implementation and performance evaluation of an adaptable failure detector*, Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002), 2002, pp. 354–363.

[BO83] Michael Ben-Or, *Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols*, Proceedings of the second annual ACM symposium on Principles of distributed computing, 1983, pp. 27–30.

[BR94] Kenneth P. Birman and Robbert Van Renesse, *Reliable distributed computing with the isis toolkit*, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[BWKI02] Claudio Basile, Keith Whisnant, Zbigniew Kalbarczyk, and Ravi Iyer, *Loose synchronization of multithreaded replicas*, SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02) (Washington, DC, USA), IEEE Computer Society, 2002, pp. 250–255.

[BWSS05] Bela Ban, Ben Wang, Manik Surtani, and Brian Standsberry, *TreeCache: a tree structured replicated transactional cache*, JBoss Inc., 2005.

[Cas01] Miguel Castro, *Practical Byzantine fault-tolerance*, Ph.D. thesis, Massachusetts Institute of Technology, 2001.

[CDK94] George Coulouris, Jean Dollimore, and Tim Kindberg, *Distributed systems: Concepts and design*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[CDK$^+$02] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana, *Unraveling the web services web: An introduction to soap, wsdl, and uddi*, IEEE Internet Computing **6** (2002), no. 2, 86–93.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup, *Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography (extended abstract)*, Symposium on Principles of Distributed Computing, 2000, pp. 123–132.

[CL99] Miguel Castro and Barbara Liskov, *Practical Byzantine fault tolerance*, OSDI '99: Proceedings of the third Symposium on Operating Systems Design and Implementation, USENIX Association, 1999, pp. 173–186.

[CM84] Jo-Mei Chang and N.F. Maxemchuck, *Reliable broadcast protocols*, ACM Transactions on Computer Systems **2** (1984), no. 3, 251–273.

[CT96] Tushar Deepak Chandra and Sam Toueg, *Unreliable failure detectors for reliable distributed systems*, Journal of the ACM **43** (1996), no. 2, 225–267.

[DHRK06] Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Rüdiger Kapitza, *Deterministic multithreading for Java-based replicated objects*, Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'06, Dallas, Texas, Nov 13-15, 2006), 2006, pp. 516–521.

[DKK$^+$01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, *Wide-area cooperative storage with CFS*, Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01) (Chateau Lake Louise, Banff, Canada), October 2001, pp. 202–215.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer, *Consensus in the presence of partial synchrony*, J. ACM **35** (1988), no. 2, 288–323.

[DN66] Ole-Johan Dahl and Kristen Nygaard, *SIMULA: an ALGOL-based simulation language*, Commun. ACM **9** (1966), no. 9, 671–678.

[DSS98] Xavier Défago, Andre Schiper, and Nicole Sergent, *Semi-passive replication*, SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (Washington, DC, USA), IEEE Computer Society, 1998, pp. 43–50.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán, *Total order broadcast and multicast algorithms: Taxonomy and survey*, ACM Comput. Surv. **36** (2004), no. 4, 372–421.

[FDES99] Pascal Felber, Xavier Défago, Patrick Eugster, and André Schiper, *Replicating CORBA objects: a marriage between active and passive replication*, Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99) (Helsinki, Finland), June 1999, pp. 375–387.

[Fel98] Pascal Felber, *The CORBA object group service: A service approach to object groups in CORBA*, Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998, Number 1867.

[FGS98] Pascal Felber, Rachid Guerraoui, and Andre Schiper, *The implementation of a CORBA object group service*, Theory and Practice of Object Systems **4** (1998), no. 2, 93–105.

[FH02] Roy Friedman and Erez Hadad, *FTS: A high-performance CORBA fault-tolerance service*, Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002, pp. 61–68.

[FHS04] Faith Ellen Fich, Danny Hendler, and Nir Shavit, *On the inherent weakness of conditional synchronization primitives*, Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004, AM, 2004, pp. 80–87.

[FJRS01] Pascal Felber, Ben Jai, Rajeev Rastogi, and Mark Smith, *Using semantic knowledge of distributed objects to increase reliability and availability*, WORDS '01: Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01) (Washington, DC, USA), IEEE Computer Society, 2001, pp. 153–160.

[FK03] Roy Friedman and Alon Kama, *Transparent fault tolerant Java virtual machine*, SRDS '03: Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS'03) (Washington, DC, USA), IEEE Computer Society, 2003, pp. 319–328.

[FLP85] Michael J. Fischer, Nancy Lynch, and Michal S. Paterson, *Impossibility of distributed consensus with one faulty processor*, Journal of the ACM **32** (1985), no. 2, 374–382.

[FMLF97] Joni Fraga, Carlos A. Maziero, Lau Cheuk Lung, and Orlando G. Loques Filho, *Implementing replicated services in open systems using a reflective approach*, ISADS '97: Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems (Washington, DC, USA), IEEE Computer Society, 1997, pp. 273–28–2822.

[FR03] Marc Fleury and Francisco Reverbel, *The JBoss extensible server*, Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference (Markus Endler and Douglas Schmidt, eds.), LNCS, vol. 2672, Springer-Verlag, 2003, pp. 344–373.

[Gar05] Jesse James Garrett, *Ajax: A new approach to web applications*, Adaptive Path LLC, Feb. 2005, http://www.adaptivepath.com/publications/essays/archives/000385.php (valid 2006-09-16), 2005.

[Gif79] David K. Gifford, *Weighted voting for replicated data*, SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1979, pp. 150–162.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *Java language specification, third edition*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[GMAB$^+$83] Hector Garcia-Molina, Tim Allen, Barbara T. Blaustein, R. Mark Chilenskas, and Daniel R. Ries, *Data-patch: Integrating inconsistent copies of a database after a partition.*, Symposium on Reliability in Distributed Software and Database Systems, 1983, pp. 38–44.

[Hay98] Mark Hayden, *The Ensemble system*, Ph.D. thesis, Cornell University, Computer Science, TR98-1662, 1998.

[HBG$^+$01] Franz J. Hauck, Ulrich Becker, Martin Geier, Erich Meier, Uwe Rastofer, and Martin Steckermeier, *AspectIX: a quality-aware, object-based middleware architecture*, Proceedings of the 3rd Int. Conf. on Distributed Applications and Interoperable Systems, 2001, pp. 115–120.

[HKRS05] Franz J. Hauck, Rüdiger Kapitza, Hans P. Reiser, and Andreas I. Schmied, *A flexible and extensible object middleware: CORBA and beyond*, Proceedings of the Fifth International Workshop on Software Engineering and Middleware, ACM Digital Library, 2005.

[Hoa74] Charles Antony Richard Hoare, *Monitors: An operating system structuring concept.*, Commun. ACM **17** (1974), no. 10, 549–557.

[HvDvS$^+$95] Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge, *An object model for flexible distributed systems*, Proceedings of the 1st Annual ASCI Conference, 1995, pp. 69–78.

[HW91] Hans-Ulrich Heiss and Roger Wagner, *Adaptive load control in transaction processing systems*, VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1991, pp. 47–54.

[JPPMA00] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Sergio Arévalo, *Deterministic scheduling for transactional multithreaded replicas*, SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00) (Washington, DC, USA), IEEE Computer Society, 2000, pp. 164–173.

[KA98] Bettina Kemme and Gustavo Alonso, *A suite of database replication protocols based on group communication primitives*, ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems (Washington, DC, USA), IEEE Computer Society, 1998, pp. 156–163.

[Kay93] Alan C. Kay, *The early history of Smalltalk*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages (New York, NY, USA), ACM Press, 1993, pp. 69–95.

[Kaz88] Michael L. Kazar, *Synchronization and caching issues in the andrew file system.*, USENIX Winter, 1988, pp. 27–36.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao, *OceanStore: An architecture for global-scale persistent storage*, Proceedings of ACM ASPLOS, ACM, November 2000, pp. 190–201.

[KC03] Jeffrey O. Kephart and David M. Chess, *The vision of autonomic computing*, Computer **36** (2003), no. 1, 41–50.

[KDH$^+$06] Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt, *FORMI: Integrating adaptive fragmented objects into Java RMI*, IEEE Distributed Systems Online, vol. 7, no. 10, 2006, art. no. 0610-o10001, 2006.

[KDK$^+$89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger, *Distributed fault-tolerant real-time systems — the Mars approach*, IEEE Micro **9** (1989), no. 1, 25–40.

[KHR04] Rüdiger Kapitza, Franz J. Hauck, and Hans P. Reiser, *Decentralized, adaptive services: The AspectIX approach for a flexible and secure grid environment*, Proceedings of the GSEM 2004 Conferences (GSEM, Erfurt, Germany, Nov., 2004), LNCS 3270, Springer, 2004, pp. 107–118.

[KKSH05] Rüdiger Kapitza, Michael Kirstein, Holger Schmidt, and Franz J. Hauck, *FORMI: an RMI extension for adaptive applications*, Proceedings of the 4th Workshop on Adaptive and Reflective Middleware, 2005.

[KMMS98] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith, *The SecureRing protocols for securing group communication*, Proceedings of the 31st Annual Hawaii International Conference on

System Sciences (HICSS), vol. 3, IEEE Computer Society Press, 1998, pp. 317–326.

[KRH05] Rüdiger Kapitza, Hans P. Reiser, and Franz J. Hauck, *Stable, time-bound references in context of dynamically changing environments*, MDC'05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems - Workshops (ICDCS 2005 Workshops), 2005, pp. 603–609.

[KSH05] Rüdiger Kapitza, Holger Schmidt, and Franz J. Hauck, *Platform-Independent Object Migration in CORBA*, On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE, LNCS 3760, Springer Verlag, Oct 2005, pp. 900–917.

[KSS96] Steve Kleiman, Devang Shah, and Bart Smaalders, *Programming with threads*, SunSoft Press, Mountain View, CA, USA, 1996.

[Lam78] Leslie Lamport, *The implementation of reliable distributed multiprocess systems*, Computer Networks (1976) **2** (1978), no. 2, 95–114.

[Lam87] ______, *Distribution*, Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87, 1987, http://research.microsoft.com/users/lamport/pubs/distributed-system.txt.

[Lam89] ______, *The part-time parliament*, Tech. Report 49, System Research Center, Digital Equipment Corp., Palo Alto, September 1989.

[LGG$^+$91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams, *Replication in the Harp file system*, Proceedings of 13th ACM Symposium on Operating Systems Principles, Association for Computing Machinery SIGOPS, 1991, pp. 226–38.

[LNP$^+$03] Ronald M. Levy, Jay Nagarajarao, Giovanni Pacifici, Mike Spreitzer, Asser N. Tantawi, and Alaa Youssef, *Performance management for cluster based web services*, Integrated Network Management, IFIP Conference Proceedings, vol. 246, Kluwer, 2003, pp. 247–261.

[Lyn96] Nancy A. Lynch, *Distributed algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Maf95] Silvano Maffeis, *Adding group communication and fault-tolerance to CORBA*, Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX, 1995, pp. 135–146.

[MGNS94] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro, *Fragmented objects for distributed abstractions*, Readings in distributed computing systems (T. L. Casavant and M. Singhal, eds.), IEEE Computer Society Press, 1994, pp. 170–186.

[MMSN98] Louise E. Moser, P. M. Melliar-Smith, and Priya Narasimhan, *Consistent object replication in the Eternal system*, Theor. Pract. Object Syst. **4** (1998), no. 2, 81–92.

[Mon99] Alberto Montresor, *The Jgroup reliable distributed object model*, Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Systems (DAIS '99) (Helsinki), June 1999, pp. 389–402.

[MPL92] C. Mohan, Hamid Pirahesh, and Raymond Lorie, *Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions*, Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992 (Michael Stonebraker, ed.), ACM Press, 1992, pp. 124–133.

[MR00] Achour Mostefaoui and Michel Raynal, *Low-cost consensus-based atomic broadcast*, PRDC '00: Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing (Washington, DC, USA), IEEE Computer Society, 2000, pp. 45–52.

[MS88] Keith Marzullo and Frank Schmuck, *Supplying high availability with a standard network file system*, Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS) (Washington, DC), IEEE Computer Society, 1988, pp. 447–455.

[MS96] Maged M. Michael and Michael L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA), ACM Press, 1996, pp. 267–275.

[MW92] Axel Mönkeberg and Gerhard Weikum, *Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing*, VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1992, pp. 432–443.

[MWN02] Scott McLean, Kim Williams, and James Naftel, *Microsoft .NET remoting*, Microsoft Press, Redmond, WA, USA, 2002.

[NAV03] Jeff Napper, Lorenzo Alvisi, and Harrick Vin, *A fault-tolerant Java virtual machine*, Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003), DCC Symposium, June 2003, pp. 425–434.

[NGYS00] Balachandran Natarajan, Aniruddha S. Gokhale, Shalini Yajnik, and Douglas C. Schmidt, *DOORS: Towards high-performance fault tolerant CORBA*, International Symposium on Distributed Objects and Applications, 2000, pp. 39–48.

[NKMMS99] Priya Narasimhan, Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith, *Providing support for survivable CORBA* *applications with the Immune system*, International Conference on Distributed Computing Systems, 1999, pp. 507–516.

[NMMS97] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith, *Exploiting the internet inter-ORB protocol interface to provide CORBA with fault tolerance*, Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, 1997, pp. 81–90.

[NMMS99] ———, *Enforcing determinism for the consistent replication of multithreaded corba applications*, SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (Washington, DC, USA), IEEE Computer Society, 1999, pp. 263–273.

[NMMS00] Nitya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith, *Transparent consistent replication of Java RMI objects.*, DOA, 2000, pp. 17–26.

[OMG01] OMG, *The common object request broker: Architecture and specification, revision 2.5*, Object Management Group (OMG) document formal/01-09-01, 2001.

[OMG02] ———, *Life cycle service specification, version 1.2*, Object Management Group (OMG) document formal/02-09-01, 2002.

[OMG04] ———, *Common object request broker architecture: Core specification, version 3.0.3*, Object Management Group (OMG) document formal/2004-03-12, 2004.

[PBWB00] Stefan Poledna, Alan Burns, Andy J. Wellings, and Peter Barrett, *Replica determinism and flexible scheduling in hard real-time dependable systems.*, IEEE Trans. Computers **49** (2000), no. 2, 100–111.

[PCD90] David Powell, Marc Chérèque, and David Drackley, *Fault-tolerance in delta-4*, EW 4: Proceedings of the 4th workshop on ACM SIGOPS European workshop (New York, NY, USA), ACM Press, 1990, pp. 1–4.

[PGJH90] Gerald J. Popek, Richard G. Guy, Thomas W. Page Jr., and John S. Heidemann, *Replication in Ficus distributed file systems.*, Workshop on the Management of Replicated Data, 1990, pp. 5–10.

[PGS98] Fernando Pedone, Rachid Guerraoui, and André Schiper, *Exploiting atomic broadcast in replicated databases*, Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing, 1998, pp. 513–520.

[PW95] David Peleg and Avishai Wool, *Crumbling walls: A class of practical and efficient quorum systems*, 14th ACM Symposium on Principles of Distributed Computing, 1995, pp. 120–129.

[RBC$^+$03] 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** (2003), no. 1, 31–50.

[RBH05] Hans P. Reiser, Udo Bartlang, and Franz J. Hauck, *A reconfigurable system architecture for consensus-based group communication*, Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS, Phoenix, AZ, Nov. 14-16, 2005), IASTED, 2005, pp. 680–686.

[RD01a] Antony Rowstron and Peter Druschel, *Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility*, 18th ACM Symposium on Operating Systems Principles (SOSP'01), October 2001, pp. 188–201.

[RD01b] Antony I. T. Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (London, UK), Springer-Verlag, 2001, pp. 329–350.

[RDH05] Hans P. Reiser, Michael J. Danel, and Franz J. Hauck, *A flexible replication framework for scalable and reliable .NET services*, Proceedings of the IADIS International Conference Applied Computing 2005, Vol I, Algarve, P, 2005, pp. 161–169.

[Rei94] Michael K. Reiter, *Secure agreement protocols: Reliable and atomic group multicast in rampart*, ACM Conference on Computer and Communications Security, 1994, pp. 68–80.

[RHD$^+$06] Hans P. Reiser, Franz J. Hauck, Jörg Domaschka, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat, *Consistent replication of multithreaded distributed objects*, SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems (SRDS'06), 2006, pp. 257–266.

[RHKS03] Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Andreas I. Schmied, *Integrating fragmented objects into a CORBA environment*, Proceedings of the Net.ObjectDays (Erfurt, Germany), 2003, pp. 264–272.

[RKDH06] Hans P. Reiser, Rüdiger Kapitza, Jörg Domaschka, and Franz J. Hauck, *Fault-tolerant replication based on fragmented objects*, Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - DAIS 2006 (Bologna, Italy, June 14-16, 2006), LNCS 4025, 2006, pp. 256–271.

[RR00] Luís Rodrigues and Michel Raynal, *Atomic broadcast in asynchronous crash-recovery distributed systems*, ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000) (Washington, DC, USA), IEEE Computer Society, 2000, pp. 288–295.

[RSH01] Hans Reiser, Martin Steckermeier, and Franz Hauck, *IDLflex: a flexible and generic compiler for CORBA IDL*, Proceedings of the Net.ObjectDays (Erfurt, Germany), 2001, pp. 151–160.

[RV92] Luís Rodrigues and Paulo Veríssimo, *xAMp: A multi-primitive group communications service.*, Symposium on Reliable Distributed Systems, 1992, pp. 112–121.

[SFGS04] Michael Schöttner, Stefan Frenz, Ralph Göckelmann, and Peter Schulthess, *Fault tolerance in a DSM cluster operating system.*, ARCS 2004 - Organic and Pervasive Computing, Workshops Proceedings, March 26, 2004, Augsburg, Germany, LNI, vol. 41, GI, 2004, pp. 44–53.

[SKK$^+$90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere, *Coda: A highly available file system for a distributed workstation environment*, IEEE Transactions on Computers **39** (1990), no. 4, 447–459.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications, ACM Press, 2001, pp. 149–160.

[SMN$^+$02] David Sames, Brian Matt, Brian Niebuhr, Gregg Tally, Brent Whitmore, and David E. Bakken, *Developing a heterogeneous intrusion tolerant CORBA system*, DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks (Washington, DC, USA), IEEE Computer Society, 2002, pp. 239–248.

[SN04] Joseph G. Slember and Priya Narasimhan, *Using program analysis to identify and compensate for nondeterminism in fault-tolerant, replicated systems*, SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (Los Alamitos, CA, USA), IEEE Computer Society, 2004, pp. 251–263.

[SN06] Joseph Slember and Priya Narasimhan, *Nondeterminism in ORBs: The perception and the reality*, DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications (Washington, DC, USA), IEEE Computer Society, 2006, pp. 379–384.

[Spe82] Alfred Z. Spector, *Performing remote operations efficiently on a local computer network*, Commun. ACM **25** (1982), no. 4, 246–260.

[Sto79] Michael Stonebraker, *Concurrency control and consistency of multiple copies of data in distributed INGRES*, IEEE Trans. Software Eng. **5** (1979), no. 3, 188–194.

[Sun04] Sun Microsystems, *Java remote method invocation specification*, Sun Technical Document, Revision 1.10, http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf (2005-02-17), 2004.

[Tho79] Robert H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Trans. Database Syst. **4** (1979), no. 2, 180–209.

[TML97] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, *Frangipani: A scalable distributed file system*, Symposium on Operating Systems Principles, 1997, pp. 224–237.

[Uy02] Edward Uy, *IP-based file system replication for Web applications*, Quest Software White Paper, http://www.quest.com/whitepapers/FS_Web_Fin.pdf (2005-12-16), 2002.

[VB98] Alexey Vaysburd and Ken Birman, *The Maestro approach to building reliable interoperable distributed applications with multiple execution styles*, Theor. Pract. Object Syst. **4** (1998), no. 2, 71–80.

[Weg90] Peter Wegner, *Concepts and paradigms of object-oriented programming*, SIGPLAN OOPS Mess. **1** (1990), no. 1, 7–87.

[Wie02] Matthias Wiesmann, *Group communications and database replication: Techniques, issues and performance*, Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002, Number 2577.

[WJH97] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang, *An adaptive data replication algorithm*, ACM Trans. Database Syst. **22** (1997), no. 2, 255–314.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, *The LOCUS distributed operating system*, SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles (New York, NY, USA), ACM Press, 1983, pp. 49–70.

[Zem06] Thomas Zeman, *Design und Implementierung einer Peer-to-Peer basierten Protokollschicht zum Zustandstransfer bei aktiver Replikation*, Diploma Thesis, Department of Distributed Systems and Operating Systems, University of Erlangen-Nürnberg, Germany, 2006.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D Joseph, and John D. Kubiatowicz, *Tapestry: A resilient global-scale overlay for service deployment*, IEEE Journal on Selected Areas in Communications **22** (2004), no. 1, 41–53.

[Zie04] Piotr Zielinski, *Paxos at war*, Tech. Report UCAM-CL-TR-593, Computer Laboratory, University of Cambridge, 2004.

[ZMMS05] Wenbing Zhao, Louise E. Moser, and P. M. Melliar-Smith, *Deterministic scheduling for multithreaded replicas*, WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (Washington, DC, USA), IEEE Computer Society, 2005, pp. 74–81.

Publications of Hans P. Reiser

1. Hans Reiser, Martin Steckermeier, and Franz Hauck. IDLflex: a flexible and generic compiler for CORBA IDL. In *Proceedings of the Net.ObjectDays (Erfurt, Germany)*, pages 151–160, 2001.

2. Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Andreas I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proceedings of the Net.ObjectDays (Erfurt, Germany)*, pages 264–272, 2003.

3. Pedro J. Clemente, Miguel A. Pérez, Sergio Luján-Mora, and Hans P. Reiser. 13th Workshop for PhD students in object oriented programming. In *ECOOP 2003 Workshop Reader (ECOOP, Darmstadt, Germany, July 21-25, 2003)*, pages 50–61. LNCS 3013, Springer, 2004.

4. Rüdiger Kapitza, Franz J. Hauck, and Hans P. Reiser. Decentralized, adaptive services: The AspectIX approach for a flexible and secure grid environment. In *Proceedings of the GSEM 2004 Conferences (GSEM, Erfurt, Germany, Nov., 2004)*, pages 107–118. LNCS 3270, Springer, 2004.

5. Hans P. Reiser, Michael J. Danel, and Franz J. Hauck. A flexible replication framework for scalable and reliable .NET services. In *Proceedings of the IADIS International Conference Applied Computing 2005, Vol I, Algarve, P*, pages 161–169, 2005.

6. Franz J. Hauck, Rüdiger Kapitza, Hans P. Reiser, and Andreas I. Schmied. A flexible and extensible object middleware: CORBA and beyond. In *Proceedings of the Fifth International Workshop on Software Engineering and Middleware*. ACM Digital Library, 2005.

7. Hans P. Reiser, Udo Bartlang, and Franz J. Hauck. A reconfigurable system architecture for consensus-based group communication. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS, Phoenix, AZ, Nov. 14-16, 2005)*, pages 680–686. IASTED, 2005.

8. Rüdiger Kapitza, Hans P. Reiser, and Franz J. Hauck. Stable, time-bound references in context of dynamically changing environments. In *MDC'05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems - Workshops (ICDCS 2005 Workshops)*, pages 603–609, 2005.

9. Hans P. Reiser, Rüdiger Kapitza, Jörg Domaschka, and Franz J. Hauck. Fault-tolerant replication based on fragmented objects. In *Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - DAIS 2006 (Bologna, Italy, June 14-16, 2006), LNCS 4025*, pages 256–271, 2006.

10. Hans P. Reiser, Franz J. Hauck, Jörg Domaschka, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 257–266, 2006.

11. Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Rüdiger Kapitza. Deterministic multithreading for Java-based replicated objects. In *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'06, Dallas, Texas, Nov 13-15, 2006)*, pages 516–521, 2006.

12. Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Hypervisor-based redundant execution on a single physical host. In *Proceedings of the 6th European Dependable Computing Conference, Supplemental Volume - EDCC'06 (Oct 18-20, 2006, Coimbra, Portugal)*, pages 67–68, 2006.

13. Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt. FORMI: Integrating adaptive fragmented objects into Java RMI. In *IEEE Distributed Systems Online, vol. 7, no. 10, 2006, art. no. 0610-o10001*, 2006.

Index