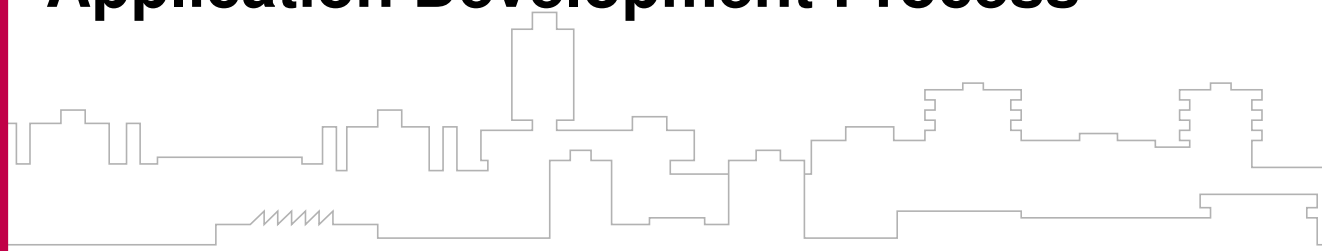
Hans Reiser

Malicious Fault Tolerance: From Theoretical Algorithms to an Efficient Application Development Process



Technical Report TR-I4-02-02
2002-04-08

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Informatik 4 (Distributed Systems and Operating Systems)
Prof. Dr. Fridolin Hofmann

Submitted to the PhDOOS workshop at ECOOP 2002, Málaga

Malicious Fault Tolerance: From Theoretical Algorithms to an Efficient Application Development Process

(Extended Abstract)

Hans Reiser (reiser@cs.fau.de)
Department of Distributed Systems and Operating Systems
University of Erlangen-Nürnberg, Germany

Abstract. In many situations, fault tolerance needs to be provided not only in the presence of fail-stop faults, but also in case of malicious misbehaviour. Recent research has provided several theoretically well-founded algorithms that are feasible in practice. Most work however focuses only on single algorithms, and gives only little attention to adaptability to different quality-of-service requirements and the whole software development process.

This thesis outline aims at making three major contributions: First, it specifies a modular architecture for malicious fault-tolerant consensus algorithms, providing a generic interface to upper layers, including recovery mechanisms, and supporting switching between different consensus strategies depending on QoS requirements. Second, it presents different abstractions for the application developer, analysing which abstraction fits best for which developer requirements, and how they can be realized using the low-level modules. Third, it discusses how the application development process for malicious fault-tolerant applications may benefit from a generative approach, using a flexible, evolvable software generation and transformation process.

1 Introduction

Tolerating benign ("fail-stop") faults in distributed systems is a well-understood matter [1]. In many situations however, the common fail-stop abstraction of faults is not adequate. Hardware may fail in a non-predictable way causing arbitrary actions before the system is halted. Software is hardly ever free of bugs and may exhibit unspecified behaviour in an unpredictable way. Last but not least, attacks to exploit security holes of current systems, especially of Internet-based systems, are so widespread that they are a significant source of malicious corruption of nodes in distributed applications.

The goal of this thesis outline is to propose a novel architecture and software development process that adequately integrates the handling of malicious faults. Particular emphasis is set on the design of the basic modular architecture for distributed consensus. Several of the proposals made in this extended abstract currently lack a final proof-of-concept by means of an implementation and sample applications. Therefore they should be considered as a visionary future system, with some details still subject to subsequent refinement.

This paper is structured as follows. Section 2 discusses the primary related work on malicious fault tolerance and specifies the points where this thesis intends to offer an improvement. Section 3 presents a generic, modular architecture that allows the dynamic selection of various consensus algorithms for different quality-of-service (QoS) requirements. Section 4 briefly addresses the question of choosing an appropriate interface for application developers. Section 5 introduces evolvable software generation techniques to further improve the application development process. Section 6 concludes.

2 Related Work

2.1 Theory and History of Fault Tolerance

Fault tolerance requires consistent redundancy in some set of nodes of the system. This is achieved by distributed consensus algorithms, which allow agreement on a common state, on coordinated actions, or on received messages, etc., in spite of fail-stop faults or malicious misbehaviour of some nodes.

The famous impossibility proof of Fischer, Lynch and Paterson shows that it is impossible to solve consensus with asynchronous communication even if only one processor fails [6]. Most subsequent research concentrated on algorithms for fully synchronous systems. This is an infeasible approach for most distributed application, e.g., those based on the Internet, as the communication medium does not provide sufficient synchrony guarantees. As a consequence, such algorithms are of little practical use.

Chandra and Toueg showed that asynchronous consensus is solvable using unreliable failure detectors [4, 5], and deduced the minimal properties such a failure detector must provide to make the problem solvable. In addition, their algorithm does not compromise the consistency of the system if those properties are not met, but only progress is impeded. Another approach to circumvent the FLP restriction is to use randomised algorithms, as will be discussed below. Such approach can solve consensus eventually with probability one.

Practical algorithms for distributed consensus may be classified into two groups: First, those that only handle fail-stop faults of some nodes (discussed in Section 2.2). Second, those that handle arbitrary misbehaviour, which often is called "Byzantine faults", especially in theory-oriented work (discussed in Section 2.3).

2.2 Handling Fail-Stop Faults in an Asynchronous Environment

The PAXOS algorithm by Lamport, one of the early publications on distributed consensus, gives an efficient solution to solve consensus in an asynchronous, fail-stop setting. Solutions for details like dynamic membership changes and implementation issues are sketched briefly [8]. Similar properties are provided by the fault-tolerant group communication protocol stack implemented by Birman et al. in the Horus [12] and Ensemble [11] projects. The main additions to PAXOS are modularity and run-time reconfigurability, while still maintaining reasonable efficiency and a complete verification. It thus shares some of the goals of this work, however in contrast to it, only deals with fail-stop faults and does not address supporting application development beyond a low-level API.

2.3 Practical Asynchronous Algorithms for Byzantine Faults

Only few projects aim at providing fault tolerance for malicious faults. An extensive work in this direction is the PhD thesis of M. Castro [3], which provides a deterministic algo-

rithm similar to PAXOS that is able to tolerate Byzantine faults, maintaining consistency in an asynchronous setting, requiring timeliness properties for progress. It also contains a complete correctness proof and is validated by a real implementation. Missing features are dynamic membership changes and run-time reconfiguration for different QoS requirements like number of tolerable faults or malicious vs. fail-stop fault tolerance. Those are significant contributions that are addressed in this thesis.

M. Reiter's Rampart toolkit [10] showed that randomisation is another feasible approach to solve consensus. It is therefore advisable to consider integrating randomisation techniques into a flexible, adaptable consensus layer. A more recent work of Cachin, Shoup and Kursawe in the context of the MAFTIA middleware project (discussed below) combines randomisation with modern cryptographic techniques and claims to provide a superior solution to consensus [2]. The paper itself only provides the bare algorithm with correctness proof. Integrating such algorithms into a software development framework will achieve further benefits for developing fault-tolerant applications.

2.4 Application Development for Tolerating Malicious Faults

The MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) middleware is a research project sponsorded by the EU, which intends to use the algorithm of Cachin, Shoup and Kursawe [2] for providing fault tolerance. The project is still in progress, so not all details of the final result are known. According to project publications up to now, the algorithm will be used to provide some general services for higher level applications. It probably will not provide a flexible, reconfigurable consensus service that is able to adopt to QoS requirements to the extent that is proposed in this thesis abstract.

Fault-tolerant CORBA is an official OMG (Object Management Group) standard for providing fault-tolerance functionality for CORBA applications. The standard only defines a general setup, and leaves many details to vendor implementations. As this thesis work intends to integrate fault tolerance in the CORBA compliant AspectIX middleware, it is worth considering the possibilities to provide an interface compatible to CORBA-FT, or to address easy portability of such applications. However, this is left for future examination.

3 A Flexible Architecture for Agreement Algorithms

3.1 Primary Design Issues

The goal of this work is to provide a flexible framework for developing a wide range of reliable distributed applications. Thus, one of the requirements is a highly reliable and sufficiently efficient platform. In addition to this, different applications will have varying demands regarding fault tolerance properties. Furthermore, those demands may change over the lifetime of an application, or may even be needed simultaneously for different users. E.g., just consider a CVS-like data repository that might be using a single central server, a set of servers for fail-stop tolerance, or even a malicious fault-tolerant setup, depending on the demands of particular user group. Thus the following design issues for a good architecture to support fault-tolerant application development are important:

1. **Dynamic Adoption to QoS Requirements:** The architecture enables adaption to different QoS requirements, e.g., choosing between fail-stop and malicious fault-tolerant algorithms, between deterministic and randomised algorithms, between consensus parameters influencing the number of tolerable faults and the efficiency in normal case operation and in the case of faults. For this purpose, this thesis outline

proposes a modular architecture that is run-time reconfigurable, encapsulates a set of algorithms with a common interface, and defines transition paths to switch between different configurations.

2. **Efficiency:** Modularisation easily has adverse effects on efficiency, e.g., by additional data copy operations between modules. In the design process, efficiency impacts should be carefully analysed and any unnecessary overhead should be avoided. The work on Ensemble [11] provides some good ideas on how to achieve good performance. "Efficiency" also depends on the particular situation: Many applications demand efficient operation in the fault-free case, and may accept larger overhead when failures occur. In contrast, real-time applications require timeliness independent of the occurrence of faults.

3. **Provable Correctness:** The intention of this extended abstract is to propose a real service for robust applications. Thus, the service itself needs to be highly robust, and consequently it is advisable to provide as much verification with formal methods as possible. Modularising the protocols into small, tractable parts is a good basis for this goal.

4. **Support for Recovery and Checkpointing:** The architecture needs to provide ways to recover and reintegrate individual nodes that have failed, as well as to restart the system after a complete failure. Therefore it needs support for consistent checkpointing and for state transfer for node recovery.

5. **Support for Upgrades:** Often it is desirable to provide some service not only without interruption by node failures, but also without or with only minimal interruption by software and hardware upgrades. The design of the basic platform should thus consider evolvability by gradual upgrading nodes and transferring state between different software versions.

3.2 Proposal of a Modular Architecture

In this thesis outline, modularisation is used as the central design principle to achieve dynamic configurability and provable correctness. The composition of the proposed architecture from its main modules is shown in Figure 1. As will be explained in this section, some of the modules may be further subdivided internally.

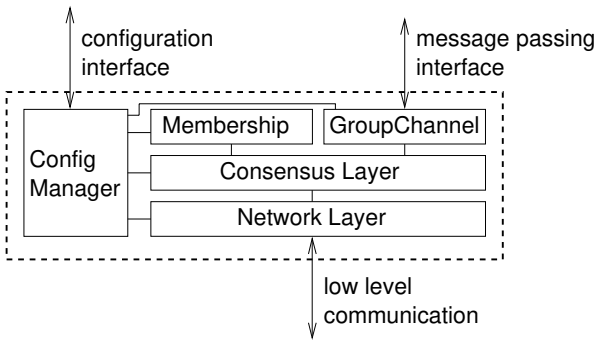


Figure 1: Modular Architecture of a Platform for Fault-Tolerant Applications

3.2.1 Network Layer

The network layer performs all necessary tasks for the low-level communication between nodes. Its main tasks are:

1. **Address Mapping:** High level names that are independent of specific communication protocols need to be translated into low level network addresses (e.g., TCP, UDP, or IP multicast address).
2. **Fragmentation:** If the low level protocol does not support messages of arbitrary size, the network layer has to provide message fragmentation and defragmentation, and optionally retransmission of lost fragments.
3. **Low Level Interface:** Mechanisms to send and receive data using selectable low-level protocols, e.g., TCP, UDP, or IP multicast, need to be provided.
4. **Security:** Some security mechanisms should be done directly in the network layer, especially to avoid denial-of-service attacks (e.g., using IPsec, packet filtering, etc.).

3.2.2 Consensus Protocol

The consensus layer provides an abstraction for various distributed consensus algorithms. It uses the network layer for communication and may optionally use the membership service to maintain group membership information. Its main task are:

1. **Group communication:** The consensus layer offers a common interface for group communication. Usually, it provides reliable, totally ordered multicast, but the exact communication semantics may depend on QoS requirements.
2. **Dynamic Reconfiguration, Checkpointing and Recovery:** The architecture must define common interfaces for replacing consensus modules, and for checkpointing and recovering their state.

As a basis for further evaluation, a first prototype implementation shall offer instances of the following types of algorithms:

- Coordination by a central node (not really fault-tolerant).
- Fail-stop fault tolerance, e.g., based on a PAXOS- or Ensemble-like algorithm.
- Deterministic malicious fault tolerance, e.g., based on Castro's work.
- Probabilistic malicious fault tolerance, e.g., based on the Cachin/Shoup/Kursawe algorithm.

3.2.3 Membership Service

The task of this service is to maintain group membership, e.g., for the group of replicas or communication partners. Different implementations are possible, like using a consensus protocol to agree on membership changes, or using an external name service to map group names to contact member names.

3.2.4 Group Channel Service

The group channel service is a high-level layer that uses membership service and consensus protocol to provide a group message delivery service with certain QoS properties. It may be accessed directly from the distributed application, or used by some upper-layer service that, for example, provides consistent active replication of objects with deterministic state.

3.2.5 Configuration Manager

The task of the configuration manager is to accept QoS requirements and configure the system accordingly. This includes selecting appropriate implementations of the components explained above and their individual configuration. In a simple setup, this can be done at system initialisation. A more powerful solution shall allow dynamic QoS changes at runtime. This task still requires further analysis. Possible configuration parameters are:

- Number of replicas
- Number of malicious faults and fail-stop faults to tolerate
- Replica consistency (strong, best effort)
- Timeliness properties
- Operation semantics (delivery and order guarantees)

4 Practical Abstractions for the Application Domain

The architecture outlined in Section 3 provides the communication infrastructure for reliable communication between functional nodes. For the application developer and for generative techniques as will be presented in Section 5, there are several options that should be considered, and implemented in another layer:

1. **Group Communication:** This service offers means for message passing with selectable QoS parameters, and is straightforward to provide using the basic communication platform.

2. **Remote Procedure Call:** RPC is a widespread abstraction used in distributed programming, and the communication infrastructure may be used to implement Multi-RPC (simultaneous RPC invocation at several nodes) with selectable semantics.

3. **Data Object Replication:** A simple form of replication is possible when we limit the objects to be pure data containers with set and get methods, but without own functionality. State changes can easily be coordinated with the basic communication platform. This results in a kind of object-oriented distributed shared memory.

4. **General Object Replication:** In the most general case, object replication can be a very complex matter, as objects may have indeterministic behaviour or external side effects. An "automagical" solution that does everything fully transparent for the application developer is not feasible in such case. For such cases, this thesis outline proposes the use of adaptable generative techniques, as presented in Section 5.

The choice of a particular abstraction should of course be made such to facilitate the application development as much as possible. Things should be automatised as far as possible in an efficient way. Where the application developer needs to have direct control of the system behaviour, he should have it. Where the application may benefit from explicit knowledge of the developer, this information should be used.

5 Configurable Code Generation and Transformation

In our CORBA compliant AspectIX middleware [7, 9], we have developed a weavelet-based code generation and transformation utility ("ADK"). It allows the application programmer

to configure and extend the transformation process with weavelets implemented in Java and an XML based weavelet configuration. The basic structure is shown in Figure 2.

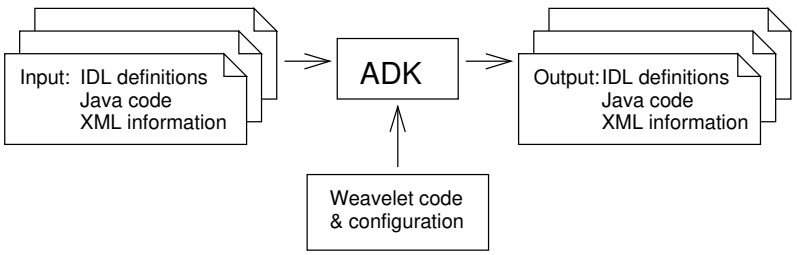


Figure 2: The AspectIX ADK

The key idea is that the developer creates a weavelet configuration for how to transform a simple implementation (e.g., a central service) into something better (e.g., a replicated, malicious fault-tolerant service). This configuration is meant to be reusable and evolvable, which means, that for a start it is sufficient to make it work for one application, and subsequent refine it when needed for further applications.

This approach seems to be optimal for fault-tolerant object replication that cannot be done fully transparent. The developer maintains full control over the code transformation process. Where possible, he still benefits from rapid development with the help of weavelet reuse.

6 Summary

This thesis outline has presented a flexible framework for the development of fault-tolerant applications. Its major contributions to the research in this area are:

- It sets its main focus on development of malicious fault-tolerant applications, which only has been addressed by few prior publications.
- The basic architecture, which has been presented in detail in this paper, has been designed for modularity and flexibility, allowing to develop systems that can adopt automatically to different QoS requirements.
- The next step will be to evaluate and implement appropriate abstractions for application developers.
- The final goal is to support application development with a toolkit based on flexible, configurable code generation and weaving.

The current state of the project is that the design specification is completed to the extent described in this paper. A prototype implementation is under development in the context of the AspectIX middleware project. A basic version of the code transformation tool described in Section 5 already exists. After completing a prototype, further evaluation of the specified design will be made.

References

[1] K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Company, Greenwich, CT, 1997.

[2] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *Symposium on Principles of Distributed Computing*, pages 123–132, 2000.

[3] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, 2001.

[4] T. D. Chandra. *Unreliable Failure Detectors for Asynchronous Distributed Systems.* PhD thesis, Cornell University, 1993.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[6] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[7] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX: a quality-aware, object-based middleware architecture. In *Proc. of the 3rd Int. Conf. on Distributed Applications and Interoperable Systems*, 2001.

[8] L. Lamport. The part-time parliament. Technical Report 49, System Research Center, Digital Equipment Corp., Palo Alto, Sept. 1989.

[9] H. Reiser, M. Steckermeier, and F. Hauck. IDLflex: a flexible and generic compiler for CORBA IDL. In *Proc. of the Net.ObjectDays (Erfurt, Germany)*, 2001.

[10] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems (LNCS 938)*, 1995.

[11] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. Ensemble security. Technical Report TR98-1703, Cornell University, Sept. 1998.

[12] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 1996.