

Norwegian University of Science and Technology (NTNU)  
Department of Computer and Information Science (IDI)

Supervisor: Svein-Olaf Hvasshovd

Techniques for Achieving  
Exactly-Once Execution Semantics  
and  
High Availability for Multi-Tier Applications

Heine Kolltveit  
<kolltvei@idi.ntnu.no>

Trondheim, 1st July 2004



# Preface

This report is submitted to the Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU) as partial fulfillment of the degree “Sivilingeniør” (MSc). The work has been carried out at the Database Systems Group.

## Acknowledgments

I would like to thank all those who gave me comments and support during my work.

My first thanks go to my advisor Professor Svein-Olaf Hvasshovd for the time he spent giving advice and comments.

Thanks to Jørgen Løland for valuable help with typesetting in  $\text{\LaTeX}$  and proofreading.

I thank the technical and administrative staff at IDI for providing the necessary infrastructure.



# Abstract

The results of this state-of-the-art report on exactly-once execution semantics and replication for multi-tier applications show that the techniques should be integrated to achieve high availability and end-to-end failure masking. This is important because unmasked failures and unavailability causes annoyance for users and lost revenue for companies.

By using transactions, consistency and at-most-once execution semantics is gained for a system. This needs to be extended to exactly-once and include explicit end-user interaction control to achieve end-to-end failure masking. Replication of processing is used to provide failure masking and to avoid system unavailability. Techniques for extending the transaction concept to achieve exactly-once with end-user interaction control, replication techniques and the replicated invocation problem presented in this report.

Although most of the fifteen state-of-the-art approaches presented in this report have support for replication and transactions, and some of them exactly-once with explicit end-user interaction control, a system that provides high availability and end-to-end failure masking including the client is still missing. A short outline for how the integration of the techniques can be done, to provide such a system, is given as a part of the analysis.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Liveness and Safety . . . . .	2
1.2	Definitions and System Model . . . . .	2
1.3	Outline . . . . .	5
<b>2</b>	<b>Exactly-Once and Failure Masking</b>	<b>7</b>
2.1	Execution Semantics . . . . .	7
2.2	Transaction Processing . . . . .	8
2.3	Exactly-Once and Failure Masking Systems . . . . .	9
2.3.1	Transactional Exactly-Once . . . . .	10
2.3.2	The Phoenix Project . . . . .	11
2.3.3	ITRA . . . . .	15
2.3.4	CORBA . . . . .	15
2.3.5	iSAGA . . . . .	18
2.3.6	Zero-Loss Web Services . . . . .	19
2.4	End-User control . . . . .	20
2.4.1	Direct Transaction Processing . . . . .	21
2.4.2	Queued Transaction Processing . . . . .	21
2.4.3	TEO . . . . .	22
2.4.4	Phoenix/APP . . . . .	22
2.4.5	ITRA . . . . .	23
2.4.6	CORBA . . . . .	23
2.4.7	iSAGA . . . . .	23
2.4.8	Zero-Loss Web Service . . . . .	24
2.4.9	Input and Output Lists . . . . .	24
<b>3</b>	<b>Replication and Replicated Invocations</b>	<b>25</b>
3.1	Replication Techniques . . . . .	26
3.1.1	Active Replication . . . . .	26
3.1.2	Passive Replication . . . . .	27
3.1.3	Semi-Active Replication . . . . .	28
3.1.4	Semi-Passive Replication . . . . .	28
3.2	Group Communication . . . . .	30

3.2.1	Atomic Multicast . . . . .	30
3.2.2	Group Membership Service . . . . .	30
3.2.3	Group Consensus . . . . .	31
3.3	Replicated Invocations . . . . .	32
3.3.1	Deterministic Execution and the Duplicate Invocation Problem . . . . .	32
3.3.2	Non-Deterministic Execution and the Orphan Invocation Problem . . . . .	33
3.4	Systems Addressing the Replicated Invocation Problem . . . . .	35
3.4.1	Deterministic Systems . . . . .	35
3.4.2	Non-Deterministic Systems . . . . .	40
<b>4</b>	<b>Analysis and Discussion</b>	<b>45</b>
4.1	Exactly-once and Transactions . . . . .	45
4.1.1	Stateful Tiers . . . . .	45
4.1.2	Execution Semantics . . . . .	47
4.1.3	End-User Interaction . . . . .	47
4.1.4	Replication . . . . .	47
4.1.5	Recovery . . . . .	48
4.2	Replication and Replicated Invocation . . . . .	48
4.2.1	Replication . . . . .	48
4.2.2	Execution . . . . .	48
4.2.3	Transaction Support . . . . .	49
4.2.4	Replicated Invocations . . . . .	50
4.3	Combining Forces to Achieve Availability . . . . .	50
4.3.1	Unmasked Failure Scenarios . . . . .	51
4.3.2	Consistent Output from Transactions . . . . .	52
4.3.3	Combining Approaches . . . . .	53
4.3.4	A Combination of Preventing Orphans and Phoenix/APP . . . . .	54
<b>5</b>	<b>Conclusions and Further Work</b>	<b>57</b>

# List of Figures

1.1	The system model . . . . .	4
2.1	Overview of <b>CORBA</b> . . . . .	16
2.2	The Object Transaction Service . . . . .	17
2.3	The server side components of Fault Tolerant CORBA . . . . .	18
3.1	Active replication . . . . .	26
3.2	Passive replication . . . . .	27
3.3	Semi-passive replication . . . . .	29
3.4	An example of a duplicate invocation . . . . .	33
3.5	A non-deterministic passive server executing a replicated invocation . . . . .	34
3.6	Non-deterministic passive replication and a primary failure lead to an orphan request . . . . .	34
3.7	A duplicated invocation . . . . .	37
3.8	Pre-filtering in <b>Symmetric Proxies</b> . . . . .	38
3.9	A write request using hush-messages in <b>Wide-Area Systems</b> . . . . .	39
3.10	Passively replicated non-deterministic server <i>C</i> invoking pessimistic server <i>D</i> in <b>Preventing Orphans</b> . . . . .	41
3.11	Passively replicated non-deterministic server <i>C</i> invoking optimistic server <i>D</i> <b>Preventing Orphans</b> . . . . .	42
3.12	Two failures at two different and passively replicated servers in <b>Reconciling</b> . . . . .	44
4.1	A failure free run of the combination of <b>Phoenix/APP</b> and <b>Preventing Orphans</b> . . . . .	54



# List of Tables

4.1	A comparison of the systems in Chapter 2 . . . . .	46
4.2	A comparison of the systems in Chapter 3 . . . . .	49



# Chapter 1

## Introduction

The computer world of today becomes more and more distributed. Computers are communicating through various networks (LANs, WLANs, WANs, etc.) over increasing distances. Thus, applications are also becoming increasingly distributed, and often rely on a chain of invocations. For instance, a user might access a web page by using a web-browser. The webpage is normally located at a remote web server and its content is generated by an application server that fetches data from a database server. Typically, all these servers reside in physically different places.

Longer invocation chains leads to a higher probability that one of the involved components will fail. Thus, the probability of a partial failure of a distributed system is higher than a failure of a centralized one. A partial failure is typically a failure of one of the servers in an invocation chain, like the one just described. If one of the servers fails, the rest can be left unable to do their job. Such a failure must be contained in the system, masked from the end-user, and all single-point-of-failures (SPOF) must be eliminated. The *only* way to deal with this, and still maintain availability, is *replication*. Unfortunately, replication alone does not guarantee that failures are masked. A mechanism that masks all failures and presenting the end-user with an illusion of *exactly-once* processing of requests is also needed.

Clearly, availability is important to be able to serve users at all times. Nevertheless, for most applications, availability is of no use if the system does not return correct results. The system must consequently be kept in a consistent state, and side-effects of replication must be eliminated. Side-effects are typically caused by partially complete operations caused by failures, and if allowed to survive they jeopardize the consistency of the system. By enclosing operations in *transactions*, problems like these do not have to be solved in an ad-hoc manner.

In this report, various systems and approaches are presented. Each addresses one or more of the following assertions:

- Failures must be masked,

- the system must stay consistent, and
- the system must be available.

All of these is closely related to fault tolerance and dependability [TS01, Chapter 7], and are important requirements for distributed systems.

The purpose of this report is to give an introduction to replication, transactions and exactly-once execution semantics, and then compare the most important properties of various approaches.

In this chapter two important properties for distributed systems are introduced in Section 1.1, and Section 1.2 presents the system model along with some definitions used in this report. Then, an outline for the rest of this report is given in Section 1.3.

## 1.1 Liveness and Safety

Both liveness and safety are properties needed to make a system available and consistent. The first property causes a system to eventually do something good (e.g. the eventual completion of an operation and the delivery of a reply), while the latter causes a system to do nothing wrong (i.e. not leaving it in an inconsistent state) [AS85].

Transactions are a rollback mechanism. Faults are tolerated by rolling back and undo the operations that lead to the failure, keeping the data consistent. On the other hand, replication is a roll-forward mechanism. Faults are tolerated by rolling forward and continuing the process execution on another replica. Thus, transactions and replication are respectively safety and liveness focused concepts [FN02] and they are both important aspects of a fault-tolerant system.

## 1.2 Definitions and System Model

The systems discussed here mix different terminologies. To be able to compare them, it needs to be standardized. For this purpose these clarifications are used in this report:

- *User*. This is the same as an end-user and is a human using the system. It can, in principle, also be another program, system or device that does not support any kind of fault-tolerance, thus behaving (to the system) as a human.
- *Service*. Cristian [Cri91] defines a service as “*a collection of operations whose execution can be triggered by inputs from users or the passage of time. Operation executions may result in output to the user and in service state changes*”.

- *Server*. According to Cristian [Cri91] “a server implements a service without exposing to users the internal service state representation and operation implementation details. Such details are hidden from users, who need know only the externally specified service behavior”.
- *Client*. This is the program running on a machine where the user inputs the commands. There are two different types of clients:
  - *Thin*. This is a client that cannot keep a persistent state, e.g. a traditional thin-client, called a terminal, or an applet.
  - *Thick*. This is a client that can keep a persistent state. A typical example is a browser that accepts cookies, or more generally; a program that has write access to the hard disk.
- *Node*. This is a machine in a network. It is the host of one or more servers or a client.
- *End-tier*. This is where the database servers are. The database servers do not initiate contact with the other tiers. Each database can be replicated for availability and performance reasons.
- *Middle-tier(s)*. This is all the servers lying (logically) between the client and the end-tier. There can be one or more of these tiers and each tier may be replicated across more than one node. To simplify the discussion each node is assumed to work for only one tier, although in principle, it might do work for multiple tiers. The middle-tier servers are normally web and application servers.
- *Request and reply*. A request is a message sent by a client to a server or by a server to another server [CDK01, pp. 145-153]. It requests that some operation at the remote server is carried out. A reply is an answer to a request. A reply can contain one or more return values, a field indicating that the requested operation was carried out, or an error message indicating that something went wrong.
- *Invocation*. When a client sends a request to a server, the client *invokes* an operation on that server [CDK01, p. 8]. A complete interaction, from the request to the reply, between the client and server is called a *remote invocation* or just *invocation*.
- *Replicated server and replica*. In a distributed system, multiple processes can be designated to do the same job i.e. offer the same service. These processes must be coordinated in a way that ensures that each of them has the same state. Such a *process group* is called a replicated server, and each of the processes that is executing the server code is a replica of that server.

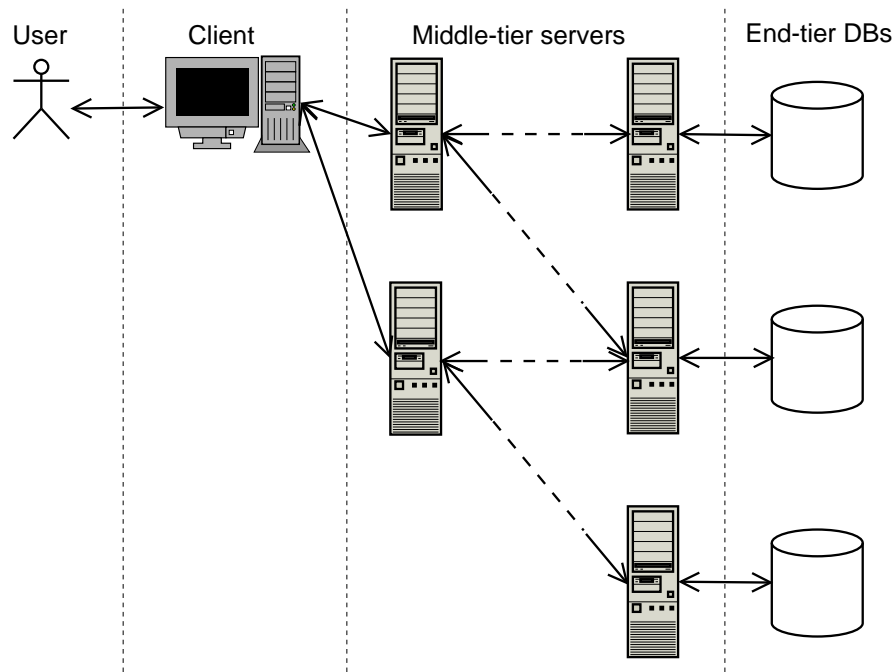


Figure 1.1: The system model

Figure 1.1 shows how the components interact. The user interacts with the client through a user interface. Typically, this involves a keyboard and a mouse for input, and a monitor for output to the user. The client can be either thick or thin, and the client interacts with one or more servers. Each server is modeled as one tier in the figure. Each tier can be replicated, but this is not shown to avoid cluttering the figure. A server can invoke the services of the other servers as if it was a client, and it can interact with the databases. Since the database servers are always transactional, the interactions with them must also be transactional, as explained in Section 2.2.

All components are assumed to be fail-stop. A fail-stop component either behaves correctly or it stops. This avoids Byzantine conditions [LSP82] that can result in erroneous replies from the servers.

In the rest of this report, the model in Figure 1.1 is used unless explicitly noted.

When a specific system or approach is presented and discussed in this report, it is usually given a name denoted by a `typewriter` font. If it is a previously named system, the name of the system is used.

## 1.3 Outline

Chapter 2 examines the exactly-once property of end-user requests and the masking of failures, without considering the availability of the systems presented. The chapter introduces the transaction processing concept, and possible extensions to it, that are needed to provide exactly-once execution semantics.

Chapter 3 presents replication as a mean to achieve fault-tolerance and availability. It looks at the most common replication techniques and the problem that occurs whenever a replicated client invokes a (replicated) server.

The various systems presented in Chapter 2 and Chapter 3 are analyzed and discussed in Chapter 4. The systems are compared and the need for combining forces to achieve a fault-tolerant service that masks failures, along with an example of how to do this, is presented.

Finally, Chapter 5 gives some concluding remarks and directions for further research.



## Chapter 2

# Exactly-Once and Failure Masking

In the last decade many techniques for achieving exactly-once execution semantics have emerged. This chapter gives a short overview of some of them. The focus is on general multi-tier architectures where there are front-end clients with human user interaction, one or more middle layers of application servers and back-end database servers. This is the same as explained in Section 1.2.

A very important aspect of the techniques is to mask all failures for the end user, and involves logically executing each request exactly once despite of multiple failures. Even if the request is actually physically executed more than once, the state-changes and reply to the end-user must be as if it was executed locally without failures. This is a stronger requirement than the traditional requirement for transaction processing system, where the system is allowed to not execute the request (e.g. abort the transaction).

First, Section 2.1 presents various execution semantics and the importance of exactly-once. Second, an introduction to transactions and the transaction processing concept is given in Section 2.2. Various techniques to render the inner parts of transaction processing systems exactly-once is presented in Section 2.3, while efforts to provide exactly-once towards the end-user is discussed in Section 2.4.

### 2.1 Execution Semantics

All failures in all tiers should be masked from the end-user. This implies that once a user has sent a request, the action(s) or operations(s) of the request should be executed exactly-once. Most systems only make a best-effort at doing this, and if something fails along the way, the user might never get a reply. To the user, this will be perceived as system unavailability, whether it is caused by a transient bug or a process crash. The systems are therefore

classified according to their *execution semantics*:

- *May-be*: Guarantees that the request will be executed no more than once. It might only execute some of the actions associated with the request.
- *At-most-once*: Guarantees that either all the actions associated with a request are done, or nothing at all is done. It is also called *all-or-nothing*, which corresponds to the atomicity property for transactions (see Section 2.2).
- *At-least-once*: Guarantees that all the actions associated with a request are done one or more times. This is typically achieved in a system that relies on at-most-once and the user manually resends requests until a reply is received.
- *Exactly-once*: Guarantees that all actions associated with a request are done, and that its effect is equivalent to a failure-free execution even in the presence of failures. This is the strongest execution semantics requirement for a system.

Notice that at-most-once also includes may-be, and that exactly-once includes all the others. If a system cannot guarantee that all failures are masked, the perceived availability of the system will be lowered. Thus, a central key to high availability is to guarantee exactly-once semantics to the user, masking all failures.

## 2.2 Transaction Processing

Transaction Processing (TP) [GR93] is a powerful failure handling concept. According to Garcia-Molina et al. [GMUW02, p. 12] a transaction is a unit of work with ACID properties<sup>1</sup>, which provides an easy way to deal with failures: If something goes wrong, the transaction is aborted by undoing all the operations it has done so far, otherwise all its work is committed.

Usually, a Transaction Manager (TM), a Log Manager (LM) and a Lock Manager<sup>2</sup> are used to support the ACID properties for concurrently executing transactions. The TM gives each transaction a unique identifier and maintains data structures of the state of each transaction. The TM also keeps track of which servers (Resource Managers, RMs) a transaction has used. This is done to facilitate the two-phase commit protocol (2PC) (for details see [GR93, pp. 562-573]), where all participants of a transaction vote

---

<sup>1</sup>Atomic, Consistent, Isolation and Durability

<sup>2</sup>The Lock Manager is out of the scope of this report, and will not be discussed any further. Interested readers are referred to [GR93].

for its outcome: Either abort it, undoing all actions done so far, or commit it, making the actions persistent and visible to other transactions.

The LM maintains the log of the system. The log is a sequential file of all the updates done to objects by transactions. The RMs tells the LM what these updates are.

The TP concept also supports recovery. The following method is the standard Undo-Redo algorithm [GR93] used by most TP-systems. When a system crashes and is restarted, all active (unfinished) transactions at the time of the crash must be aborted, and any traces of these reflected on persistent storage must be undone. The LM scans the log in a backward manner, starting at the end of the log. When it finds an update done by an unfinished transaction it is undone if its action is reflected in persistent storage, and (usually) a compensating log record (CLR) is generated. All committed interactions are reflected in the log. Changes made by these that are not on persistent storage must be redone. This is done by the LM by scanning the log from the earliest transaction which committed, but doesn't have its committed state reflected in persistent storage, and forward. This starting point can be made more recent by checkpointing.

## 2.3 Exactly-Once and Failure Masking Systems

In the previous section, the transaction processing concept was presented. Due to the possible abortion of transactions, a standard TP system offers only at-most-once execution semantics. It can be made at-least-once by users manually retrying requests until an answer is received. As this does not mask failures from users, it is not satisfactory. Therefore, a system should be able to offer exactly-once execution semantics. There are different ways to extend the TP concept to provide this:

- 2PC involving the client [LS98]. Clients participate in the commit decision of a transaction. Therefore, the client will always know the outcome of a transaction. This approach assumes that the client has stable storage. It doesn't handle the problems that might arise if a transaction coordinator crashes and all the participants remain blocked.
- Automatic retry of aborted transactions [FG99]: Clients participate in a light-weight way because they do not vote for the outcome of a transaction, but they still need the result and outcome. Exactly-once is achieved by making the middle-tier resend requests (using the same transaction identifier) until it receives an answer. Such requests must be made *idempotent*. An idempotent request has the property that the result of multiple executions of the request is exactly the same as a single execution. By using a clever identification scheme together with duplicate detection, requests can be made idempotent.

- Persistent Queues [GR93, BHM90]: This technique is based on queues in front of the server and the client. There are certain drawbacks to this approach. First, it involves the cost of three distributed commits (2PC), and second, all the servers must be stateless between transactions. The only state in the system is stored in a database or a queue. In addition, to avoid SPOF, both queues must be replicated, which incurs an added overhead and complexity of coordinating them.
- Message Logging. This approach is based on logging the interactions between the participants. The logging enables replay of the messages and therefore exactly-once execution semantic. *Phoenix/APP* (Section 2.3.2) and *ITRA* (Section 2.3.3) are examples of this technique. They both mask failures, but in two different ways: The former utilizes message logging to enable recovery, whereas the latter uses it to enable reconfiguration transparency.
- Object Groups. By using a distributed object oriented system and its services, a system can be given exactly-once execution semantics. The system must support fault-tolerant objects, usually implemented as object groups. An object contains the state of the object and code to manipulate it. Each object replica of an object group is located at different nodes to achieve fault-tolerance. A system supporting object groups, *CORBA*, is presented in Section 2.3.4.

In the following sections some systems that use these techniques are presented.

### 2.3.1 Transactional Exactly-Once

The *TEO* approach has been developed by Frølund and Guerraoui for Hewlett-Packard Laboratories [FG99]. It describes a three-tier model where the terminals issue requests to the middle-tier application servers. The application server is replicated with some form of primary-backup or active replication. The primary application server initiates transactions against the back-end database tier. If successfully it commits the transaction and delivers a reply to the client.

The client maintains a list of application servers, and if the client does not receive a reply before a time-out period a request is re-sent to all the application servers in the list. If any application server suspects<sup>3</sup> the crash of the primary, it becomes the primary and tries to complete the result. To facilitate synchronization between the servers consensus objects called write-once (wo) registers are used. This is just an abstraction of a consensus protocol (see Section 3.2.3). When an application server becomes the primary it checks the wo-register to check the status of the transactions. If

<sup>3</sup>To learn more about suspicion, see Section 3.1.4.

a transaction has committed (all the database servers have voted yes), the new primary finishes the commitment and replies to the client. Otherwise, the transaction is aborted and an error-code is returned to the client. The client can then re-issue the request.

The recovery of application servers are not modeled here. It is simply assumed that there are always enough application servers around to do the work. Also, this approach requires the clients and application servers to be stateless. Any state between transaction boundaries are therefore stored in the back-end database servers.

### 2.3.2 The Phoenix Project

The Phoenix Project is currently run by Lomet and Barga and is a project within the Microsoft Research Database Group<sup>4</sup>. The main focus is on application availability and persistence. When a database server crashes the state is recovered to the last committed transaction and all uncompleted transactions are aborted. This leaves the application in an inconsistent state which the application programmer or the operator has to deal with. The Phoenix Project uses message logging to enable the application programmer to write stateful programs, while making recovery automatic to achieve lower mean-time-to-repair (MTTR), thus increasing the availability.

The following paragraphs introduces **Phoenix/ODBC** and **Phoenix/APP**. The Phoenix Project does not cover replication and thus does not support failover.

#### **Phoenix/ODBC**

The Phoenix Project started by defining an ODBC driver (**Phoenix/ODBC**) [BLBA00] which supported persistent client-server database sessions. It survives server crashes without the client noticing, except for possible timing consideration. The ODBC driver maintains a virtual session with the database server making server failures transparent to the client. The principle can be used with any existing ODBC driver to any relational database system, and the extra overhead is modest. As this is only a two-tier architecture it will not be discussed any further in this report. Note however, that this can be used as a way to make the communication with the back-end database servers persistent and recoverable.

#### **Phoenix/APP**

**Phoenix/APP** [BLP<sup>+</sup>03] (former **Phoenix/COM** [BLW02]) is a framework for general recovery guarantees in a multi-tier environment using message logging. It defines a set of protocols, called contracts, between different kinds

---

<sup>4</sup><http://research.microsoft.com/db/>

of software components or servers. These protocols specify what needs to be logged and when the log needs to be forced to persistent storage.

There are three main types of components; persistent components (Pcoms), transactional components (Tcoms) and external components (Xcoms). A Tcom, an Xcom and a Pcom are typically a database or a transactional queue, a user and a web or application server, respectively. To make these components recoverable, all non-determinism is removed by (1) assuming that components are piecewise deterministic (PWD) and (2) logging messages. A component is PWD if it is deterministic between any two received messages from other components. This enables the component to be replayed from an earlier state to the same end state if it is fed the messages in the same order. More about determinism can be found in Chapter 3. The message logging style for the different interactions are captured by these contracts:

- A Committed Interaction Contract (CIC) or an Immediately CIC (IC-IC) between two Pcoms.
- An External Interaction Contract (XIC) between a Pcom and an Xcom.
- A Transactional Interaction Contract (TIC) between a Tcom and Pcom

The TIC and CIC can guarantee exactly-once execution semantics by logging messages. As we will see this is not necessarily true for an XIC. The following paragraphs explain the CIC, XIC and TIC, respectively.

**Persistent-Persistent Interaction** This type of interaction must guarantee that both the state and the message of each interaction are persistent. This is done using a CIC. During a committed interaction the sender of the message must keep the following obligations:

- S1: The state of the sender at the time of the sending of the message or later is persistent.
- S2a: The message is periodically resent until the receiver releases the sender from this obligation (ack-stable)
- S2b: The message is stored and resent upon explicit request from the receiver until the receiver releases the sender from this obligation (ack-installed).
- S3: The identifier of each message has unique contents.

The receiver must keep these obligations during a committed interaction:

- R1: Duplicate messages are eliminated.
- R2a: The receiver's state at the time of receiving the message or later must be persistent, without the sender periodically resending it, before an ack-stable is sent. The interaction is *stable*.

- R2b: The receiver's state at the time of receiving the message or later must be persistent, without the sender ever having to resend it, before an ack-installed is sent. The interaction is *installed*.

The obligations S2a and S2b together with S1 guarantee that an interaction is recoverable. When the interaction is installed it is recoverable without help from the sender. The reason for the distinction between S2a and S2b, and R2a and R2b is to provide a richer design space for an implementation. It makes it possible to balance the amount of message resends versus forced message logging in different environments. R1 together with S3 guarantees that if a copy of a message is received (because of S2a or a network error making a duplicate of the message) the last message is eliminated without re-execution.

The interaction does not need to be made stable (or installed) at once, but can be delayed until a reply is sent. When a reply is sent, the state of the sender (originally the receiver) must be forced to persistent storage (according to S1), so the previous interaction is automatically installed and a force write is saved.

An ICIC is the same as a CIC except that the interaction is installed as soon as it has been received. This means an extra force write to persistent storage compared to the CIC and makes the interaction synchronous.

**Persistent-External Interaction** The problem with an Xcom is that it may not be persistent, and therefore cannot have committed interactions. The (close-to) solution to this problem is using a XIC. A Pcom gives ICIC guarantees while an Xcom only gives a best effort guarantee. This leads to the following problems:

- X1: Assume that a Pcom (e.g. a client machine) sends (displays) a message to an Xcom (e.g. a human user), which includes a force of the message and state, and then crashes before it knows that the Xcom has seen the message. In this case, the Pcom has to resend the message. An Xcom might not eliminate duplicate messages and hence a user might see the same message twice.
- X2: Assume that an Xcom sends a message, via an input device, to a Pcom. The Pcom crashes before it logs the message receive. Upon restart, the user must resend the message. But an Xcom (user) has not promised to do this automatically, and only makes a best effort in doing this.

These failures are not masked and violates the exactly-once semantics. The interesting feature here is that in the failure free case during an external interaction, the result is an immediately committed interaction. This masks *all* internal failures from the Xcoms (human users).

**Persistent-Transactional Interaction** A Tcom is a component that adheres to the ACID properties of transactions and hence has an all-or-nothing state transition. The problem with this is that an interaction is not guaranteed to complete. If a transaction aborts the Tcom may forget all about it, imposing extra failure handling difficulties at the Pcom. Even if a transaction commits the final reply might not be delivered (e.g. because of a network failure). This is not adequate, and the behavior is therefore controlled by a TIC where the Tcom makes the following obligations:

- T1: The Tcom either aborts or commits the transaction, leaving only two possible (persistent) states.
- T2: The Tcom must report the commit or abort of a transaction to the Pcom. If a transaction aborts, the abort is signaled to the Pcom as a reply to the next request for the same transaction.
- T3: When the Tcom receives a commit request, it acknowledges the request through a reply message. The reply must be forced when the message is sent.

The Pcom makes the following promise:

- P1: The commit request and the state of the Pcom at the time of the request (or later) must be persistent. This also guarantees the persistence of all the earlier replies from the Tcom within the same transaction. The Pcom asks the Tcom for the final outcome of the transaction, if needed.

Let us consider the following failure cases:

- When the Pcom fails during a transaction, the Tcom forgets about the old transaction by timing out. The Pcom restarts the transaction and the Tcom replies with an “unknown transaction” (or similar) return code. The transaction is then restarted as a new transaction.
- If the Pcom fails before the commit reply message is made persistent, the Pcom queries the Tcom for the outcome during the recovery phase. If the transaction aborted, T2 guarantees that an error code (or similar) is returned to the Pcom. If it committed, T3 promises that the commit reply message persisted, so it is re-sent.
- When the Tcom fails or autonomously aborts the transaction, the Pcom must restart the transaction. The Tcom sees this as a new transaction.

The testable transaction status supported by the Tcom (in obligation T2 and T3) allows the Pcom to ask the Tcom for the outcome of a transaction and alleviate the Tcom from periodically resending the reply.

### 2.3.3 ITRA

The Inter-Tier Relationship Architecture (ITRA) has been developed by Dekel and Goft for IBM [DG01]. It uses passive replication and allows for reconfiguration of node structure while still providing services. The state of a server tier is also replicated to the proxy and the work is done in a multi-tier environment.

A tier is composed of one or more nodes for scalability and availability, and a node may “work for” more than one tier. Each tier contains the successor tiers’ stubs. A stub stores the last snapshot of the successor tier’s state and the last operations sent to the tier. When a request is passed to a stub, the stub stamps it with a unique and reproducible sequence number. The request is only forwarded to one node within the successor tier. The node is then responsible for executing it, respond with the result(s) and report back to the stub when the operation is synchronized with the server, *server synched*. An operation is server synched when its effects can be obtained from nodes within the tier without the aid of any other nodes, even in the presence of failures.

If the node fails to respond to a request, the request is sent to another node. The first node might have forwarded the request to another tier and received a reply from it again. The unique sequencing scheme guarantees that requests can be made idempotent. Idempotence is achieved by returning the same result without re-execution, if the second node makes the same request to the last tier. A log of server replies is kept until the requesting tier is *client synched*. A reply is client synched when the operation causing the reply will never have to be re-executed. Both server and client syncing is done by replicating each of the node’s states in the tier.

ITRA can be used with any kind of middleware.

### 2.3.4 CORBA

The Common Object Request Broker Architecture (CORBA) is a specification of a distributed system [TS01, chapter 9]. It has been made by the Object Management Group (OMG)<sup>5</sup> and was designed to overcome interoperability problems regarding the use of different operating systems and applications in a single system. The current release is version 3.

A basic overview of the architecture of CORBA is shown in Figure 2.1. The core of this model is the *Object Request Broker (ORB)*. It enables the communication between clients and objects, and makes the distributed and heterogeneous nature of the system transparent to the application programmers and users. The Common Object Services are the most widely used services and include services for naming, concurrency control, events, life cycling, recovery, persistence, security, etc.

---

<sup>5</sup>[www.omg.org](http://www.omg.org)

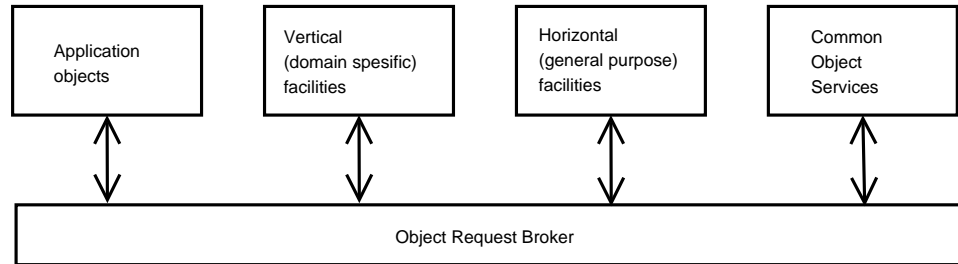


Figure 2.1: Overview of CORBA

Horizontal facilities are high-level services not related to any specific application domain. Examples include user interfaces and information, system and task management. The vertical facilities are high-level services that are targeted towards specific application domains such as banking and manufacturing. The general services, facilities and object model of CORBA is not within the scope of this report, and will therefore not be discussed any further. Instead, the focus will be on two of the services offered; transactions and fault tolerance. These are discussed in the following paragraphs.

### CORBA Object Transaction Service

The *Object Transaction Service (OTS)* [OMG03] supports flat, distributed and nested transactions. OTS provides the safety property in CORBA and uses the 2PC protocol to reliably decide the outcome of a transaction. It is initiated by a client and can involve series of invocations involving multiple objects. OTS makes distinctions between three types of objects: Recoverable, transactional and non-transactional. The first two types support transactional operations while the latter does not.

A *recoverable* object contains a state that can be changed by invocations inside a transaction. Thus, recoverable objects must be involved in the decision to commit or abort the transaction. A special *resource* object is registered in the Transaction Service to allow its involvement in the 2PC protocol. It guarantees the ACID properties by controlling e.g. transactional state changes and locks on behalf of a specific recoverable object. In particular, the resource object must ensure the temporary persistent storage of state changes as a part of the *prepare* phase of 2PC.

A *transactional* object is an object capable of being participating in a transaction, but it does not have its own state. Rather, it usually delegates the responsibility to other (recoverable) objects. Therefore, it does not need to be a part of the 2PC protocol, but it can still force a rollback of the transaction.

These objects and their interactions with the OTS are shown in Figure 2.2.

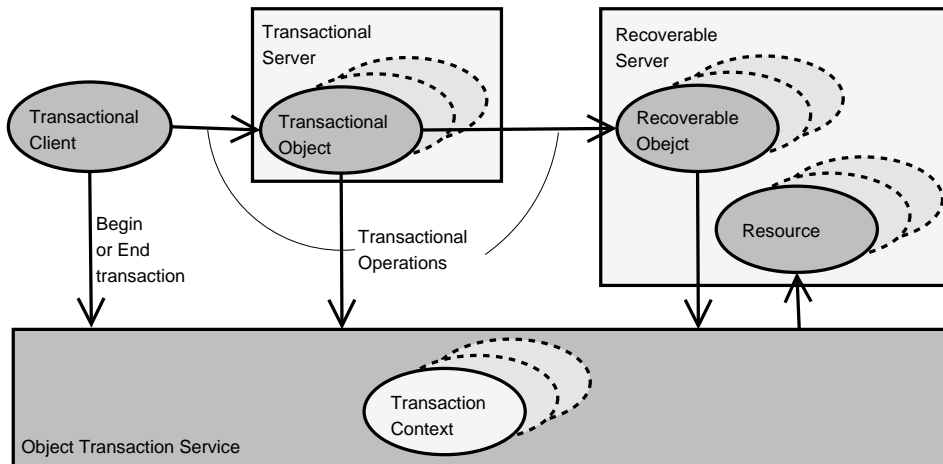


Figure 2.2: The Object Transaction Service

### Fault Tolerant CORBA

The Fault Tolerant CORBA, FT-CORBA, specification [OMG04] provides a way to support high availability and reliability, through replication, fault and recovery management. It consists of a few minimal fault tolerant services required by every FT-CORBA implementation, and policies and interfaces for more advanced management of the service. It can tolerate object-, process- and host crashes, but does not support Byzantine problems [LSP82] or network partitions. The specification allows both active and passive replication, although the execution must be deterministic.

Clients are minimal in their support for replication. When a client issues a request to a normal, non-replicated object, the object is referenced by an Interoperable Object Reference (IOR). As this reference is only for a single object, an Interoperable Object Group Reference (IOGR) is used instead if the object is replicated. A client can not see the difference between an IOR and an IOGR, but the latter includes multiple profiles, each corresponding to a replica or a gateway giving access to a replica group. The client-side ORB provides mechanisms for iterating through these profiles, and reinvoking and redirecting of requests. Together, these provide replication and failure transparency for clients.

A *replication manager*, shown in Figure 2.3, consists of a property manager, an object group manager and a generic factory, and is replicated for fault tolerance. It controls the replication and distribution of objects across the system. This involves the creation, deletion and replication of application and infrastructure objects. When the replication manager receives a message from the application, telling it to replicate a certain object, the request is distributed to host specific factories.

As Figure 2.3 shows, each host has its own fault detector that reports

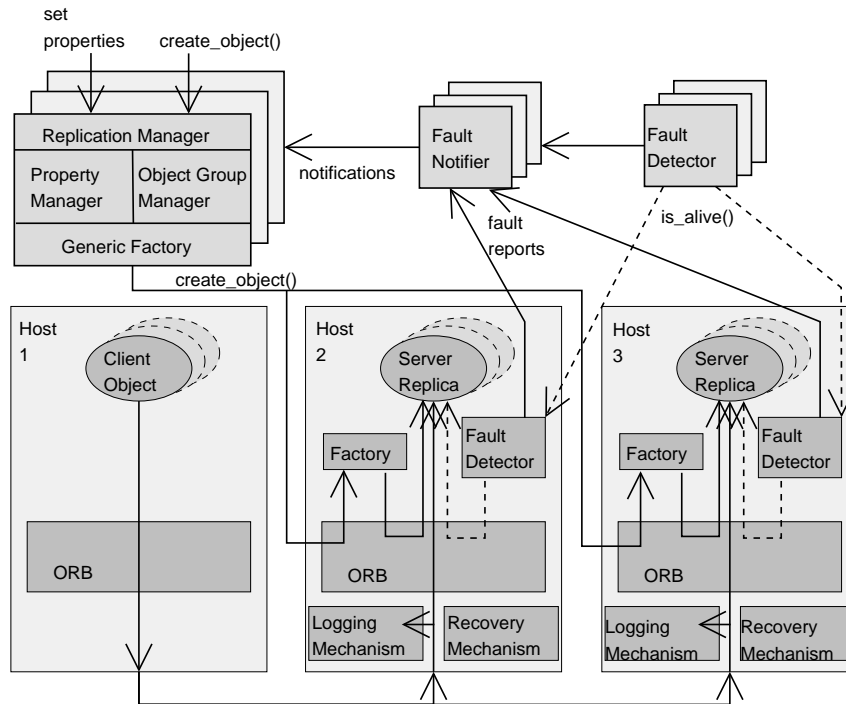


Figure 2.3: The server side components of Fault Tolerant CORBA

faults to the fault notifier. In addition, there is a replicated fault detector that periodically asks the local fault detectors if they are alive. Together they detect host, process and object faults. To recover from such faults the logging and recovery mechanisms are needed.

### 2.3.5 iSAGA

The interaction pattern between a client and server on the Internet has changed over the last few years. They started out as simple requests to download static pages. By now, they have grown into multiple requests to dynamic pages, and a lot of work can be lost because of a system failure.

An iSAGA [DVDR01] is a sequence of requests made by the user. As the user is provided with a reply (e.g. a webpage), he or she typically sends a new request (e.g. clicks on a link on that page) and receives a new reply. These interactions are modeled as long-running transaction where the user provides additional information when required. This enables the client to recover after a failure, and therefore relieves the user from having to go through the same steps all over again.

iSAGA is based on the observation that the highly constrained context of databases and the loosely constrained context of the Internet should be handled in different ways. A database system has strict concurrency control,

usually enforced by holding locks for as long as a transaction is running. Such a solution would be too much of a performance bottleneck because of slow responding users on the Internet. Therefore, while a user is thinking, his or hers view might be updated by another user or the system (e.g. a user trying to buy a seat, but the price of that seat has just been raised). This might invalidate a user's view of the system and has to be handled during recovery.

The user view might also be invalidated by the user. Typically, this happens if the user loses his or hers connection and decides to use other means of reaching the objectives (e.g. by calling the airline). By using a timeout mechanism, the user view can be invalidated, making recovery at the next reconnect unnecessary.

An iSAGA looks a lot like a long running transaction; the name "iSAGA" is in fact chosen because of the resemblance with real TP sagas [GR93, pp. 217-219]. There are, however, two important differences. Both of which are concerned with the unpredictable nature of humans:

- An iSAGA may grow to an arbitrary size, and even if it ends unexpectedly, the updates made to the underlying database will still be persistent (i.e. no compensating transactions).
- A user may change his or hers objectives along the way, and an iSAGA may therefore consist of multiple interleaving objectives. These pieces of an iSAGA, each leading to a different objective, is called a sub-iSAGA. They have to be considered during recovery.

The state of a component is saved to a state log after every request. Depending on the architecture, this log can reside on the client node, server node or even on another node. During recovery, the log is checked for previous states for the iSAGA and only those states that are a part of the latest sub-iSAGA are kept. The validity of these states is checked; invalid states are removed and the most recent valid state is then presented to the user.

### 2.3.6 Zero-Loss Web Services

In a standard clustered server approach, any ongoing requests on a failing server are lost. Since lost requests are not acceptable, a non-loss web service is needed. Such a service, **Zero-Loss Web Service** is presented by Luo and Yang [LY01]. The system model used here seems to differ from the one defined in Section 1.2 (see Figure 1.1). In this model, a dispatcher is placed between the clients and the application servers, but the dispatcher may be viewed as the first application server using other application servers. Thus, the model can still be used.

The dispatcher controls the routing of requests to the replicated servers. The request-routing mechanism is also extended with two new concepts:

- *Request discrimination.* As the logging of every incoming request is costly and might be a performance bottleneck, the administrator can decide which (types of) requests need fault-tolerance and which do not. As the non-fault tolerant case is not relevant in the scope of this report, it is not discussed any further.
- *Request migration.* The dispatcher can migrate a request from one server to another. This can be done because of a failure or just for load-balancing.

Requests fall in three different categories; requests for static or dynamic pages, or session-based requests. As the two former ones are trivial compared to the latter, only the latter one is presented here.

With session based requests, a session needs to be defined. This means that its beginning, end, and internal state have to be identified. This is done by the system administrator or programmer. When the dispatcher receives a request which corresponds to the start of a session, it recognizes it and redirects all the following requests from the same client to a primary and a backup server. The replication is synchronized in the sense that the dispatcher forwards the requests to both servers, but the backup does not execute transactions against the database tier and it does not reply to the client unless the primary fails. The backup also receives the replies from the database-tiers. Both servers are therefore guaranteed to have the same session processing state. This is a variant of the passive replication approach presented in Section 3.1.2.

The dispatcher can be replicated using a logical ring, where the non-deterministic events are checkpointed to the neighbour dispatchers to avoid causing a SPOF.

## 2.4 End-User control

The essence of exactly-once is to mask all failures from the user. This means that system must not only be able to deal with server failures, but client failures must be handled as well. Therefore, the interactions between the end-user and the client must be carefully controlled. Sadly, it seems to be impossible to achieve total control of these interactions due to the nature of users. The two following scenarios illustrate the problem:

- The user sends a request to the client and the client crashes before processing or making the request persistent. The user might need to resend such a request, causing an unmasked failure.
- The client sends a reply to the user and then crashes before it knows that the user has seen it. The client has no way of knowing what

the correct action is to preserve exactly-once execution semantics. Depending on the solution chosen the user might see duplicate messages or none at all, neither of which is consistent with exactly-once execution semantics.

These two issues need to be handled in the best possible manner. It is a question about atomicity for the interactions between a client and a user. In the presence of a client crash the system cannot verify (during recovery) whether or not an output has been seen by the user, and the user cannot verify if the client has seen its input. Also, output from a transaction should not be shown to a user before the transaction has committed or as a part of the commit. If the user was able to see an uncommitted output from a later aborted transaction, the user might act on it and jeopardize the consistency of the system.

The following subsections will present the different approaches of the systems presented above, in addition to an approach used in databases.

### 2.4.1 Direct Transaction Processing

In *direct TP systems* (**Direct TP**) all “requests are immediately authorized, dispatched, and executed, and replies are returned to the user without delay” [GR93, p. 333]. The problem with this paradigm is that the information about the transaction’s outcome might be lost. Once a transaction has committed, the Transaction Manager (TM) forgets all about it. If the presumed abort protocol<sup>6</sup> is used the TM also forgets about an aborted transaction. Therefore, if the request or the reply gets lost because of an untimely system failure, there is no way for the client to know whether the transaction was executed or not. Also, if the transaction aborts, no reply is delivered to the client. This is clearly not good enough as the user might never be notified of the transaction outcome.

### 2.4.2 Queued Transaction Processing

*Queued TP systems* (**Queued TP**) involve two recoverable queues: One in front of the server (server-queue) and another in front of the client (client-queue). A client starts a transaction and enqueues the request at the server-queue. The server starts another transaction, dequeues and processes the request, and enqueues the reply at the client-queue. A third transaction is started, and the reply is dequeued and processed by the client. The client processing is where the reply is presented to the user, and the user has to acknowledge the reply before the third transaction is committed.

All the transactions make use of the 2PC protocol to decide the outcome. If a transaction aborts, the request or reply is returned to the queue, and

---

<sup>6</sup>The presumed abort protocol is a special implementation of 2PC, mentioned earlier in this part.

is processed by a new transaction. By making the user a part of the 2PC protocol, exactly-once is achieved. The problem with this approach is that the client has to be executed on a system that supports transactions. This requires a thick-client to facilitate the need for persistent storage required by 2PC. Also, the added overhead caused by three 2PC decisions is not trivial for most applications.

Bernstein et al. [BHM90] improves the queued TP-system concept. They define protocols to ensure server processes requests exactly-once and client processes replies at-least-once. As in normal queued TP two queues, client-queue and server-queue, are used. The difference is that the need for client transactions is removed by logging non-idempotent operations along with the state of the client at the time it was issued. Sending the request to the queue is a non-idempotent operation. A server-queue can aid a recovering client by telling it whether a request was enqueued or not, and a client-queue can keep a reply for re-dequeueing until the client explicitly deletes it. To completely relieve the client from having a persistent state, the state of the client can be piggybacked on the enqueue and dequeue operations, and making the queue save the checkpoint. When the client process fails and later reconnects to the queues, the latest snapshots of the state is transferred back to the client. The user might need to see the last message again, thus the client should be able to re-request the latest reply. This enables the use of a thin-client at the loss of making reply processing at-least-once.

Each request is given a unique identifier which is visible to the user. In the presence of a client crash, this enables the user to resend the original request using the same identifier. Of course, this requires the user to remember the correct identifier. If the same identifier is supplied with the exact same request this leads to exactly-once processing as argued by the authors. Strictly, this is nothing more than a best-effort and exactly-once request processing is only achieved after the request has been successfully enqueued.

### 2.4.3 TEO

To achieve exactly-once execution semantics this approach resends requests from client to middle-tier servers until a reply is received. If the client crashes this solution can only guarantee at-most-once, and the user needs to find out what happened manually. Also, this approach does not mention the client to user interaction. Therefore, it is not discussed any further in this section.

### 2.4.4 Phoenix/APP

In *Phoenix/APP* the user is modeled as an external component (Xcom) and the client as a persistent component (Pcom). The interaction between them is captured in a protocol called XIC, which is explained in Section 2.3.2. To summarize: A request is logged as quickly as possible, and a reply is replayed

if the client fails without knowing if the user saw it.

The main reason for handling the interaction in this way is to make it persistent as soon as possible. The argument is that it is impossible to guarantee exactly-once execution semantics for these interactions. This is acknowledged by making the window of opportunity for a failure, leading to a violation of exactly-once, as small as possible. The result is at-most-once for request processing and at-least-once for reply processing. If the client does not fail during this small window, however, the effect is exactly-once request processing.

#### **2.4.5 ITRA**

ITRA issues a unique identifier with each request. This makes the retry of a request idempotent because the next tier does not re-execute a request that has already been executed. The client to user interaction is however not discussed for this approach.

#### **2.4.6 CORBA**

The focus in CORBA is on service availability for interactions without state and, although supported by the standard, no known implementation completely masks failures to users. The standard does not support client failure or migration and does not cover end-user interaction.

#### **2.4.7 iSAGA**

For iSAGA, there is no description of the user to client interaction regarding the handling of requests. The request and the state at the time of the send are captured in the logged state. Where this happens depends upon the chosen architecture and implementation. During recovery the latest valid state is reconstructed. Because this state might, or might not, be the latest state presented to the user, this means that no guarantees regarding execution semantics are given. A reply might never be seen by a user if it is lost and then invalidated, and a reply might be seen more than once if it is received and then recovered. The user is simply presented with the latest “accepted” action it performed.

Although this may still be a viable solution for Internet Web Services, it is not the case for general multi-tier architectures. A lost reply means that the user has no way of knowing if the actions associated with the request has been executed or not. Also, automatically resending the request is not idempotent as there is no mechanism to remove such request at the server side.

### 2.4.8 Zero-Loss Web Service

This approach, as described in Section 2.3.6, does not model the client to user interaction, and a request is just recoverable in the presence of a server failure. If the client fails, the user must manually find out whether the last request was carried out or not.

### 2.4.9 Input and Output Lists

Cristian [FCK87] discusses, a two-tier system, where a client application interacts directly with a database server. Although this does not fit the system model in this report (see Figure 1.1), some interesting points are made regarding user to client interaction.

The inputs from the user are logged in a list,  $i$ , and the outputs that have been sent to the terminal window are logged in another list,  $o$ . Removing a request from  $i$  and logging it at the database is an atomic action. Presenting the user with the reply and logging it in  $o$  is also an atomic action. Thus, if the client or the database crashes, the user does not need to log on again nor repeat the last inputs to the client. Neither does the client need to repeat any output to the user. In this report we refer to this approach as **Input and Output Lists**.

## Chapter 3

# Replication and Replicated Invocations

According to Murphy's Law, "anything that can go wrong, will go wrong". This also applies to computer science and is of course why fault tolerant systems were developed. Particularly, in a distributed system, if one node fails the whole system might be rendered unavailable. The solution is obvious: Introduce redundancy and replicate the servers to avoid the danger of single point of failures to take down the entire system.

Replication of a component provides a mean to continue the execution in the presence of a component failure. Although, in principle, a component might be any kind of component in a computer system, this report discusses only the replication of processes (or servers). If one replica of the server crashes, the execution is automatically resumed or continued on another replica. Ideally, this should happen automatically, making the client indifferent of the replicated nature of the server. This is called *replication transparency*.

Although replicating servers might seem straightforward, it is not. Special care needs to be taken to ensure the consistency and the availability of the system. This includes the handling of replicated invocations.

A *replicated invocation* occurs whenever a replicated server uses another (possibly replicated) server. Such an invocation must have the same outcome and lead to the same state changes as the same invocation between non-replicated servers.

This chapter starts off by giving a short overview of the most influential replication techniques in Section 3.1. Then, as each of the replication techniques requires a non-empty set of various group communication protocols, the most important ones are discussed in Section 3.2. Section 3.3 introduces and discusses the inherent problems to a replicated server invoking another server. Finally, various systems and concepts that utilize replicated invocations are presented in Section 3.4.

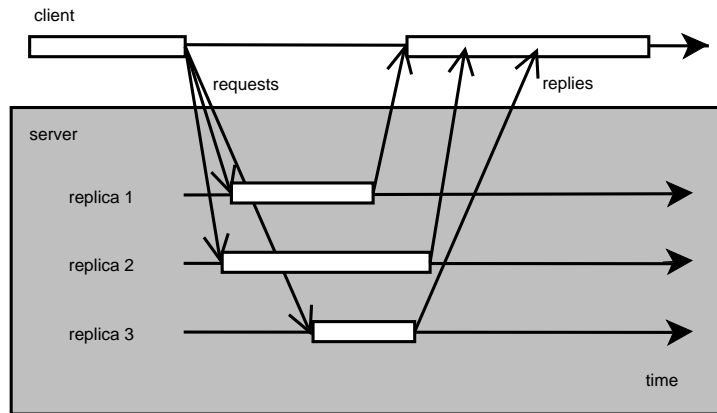


Figure 3.1: Active replication

### 3.1 Replication Techniques

Replication increases a system's availability. Different techniques or implementations of replication have been developed. All the protocols implementing them must obey the correctness criterion *linearizability* [HW90]. This is also called one-copy equivalence, because it gives the client the illusion of interacting with non-replicated servers. In practice, linearization is achieved if requests<sup>1</sup> are executed by all replicas in an ordered and atomic way.

There are two main types of protocols; *active* and *passive*. As shown in the following sections they have complementary applications and qualities. Some hybrid protocols are also presented to show that trade-offs can be made.

#### 3.1.1 Active Replication

In active replication all of the replicas perform the same work. This is also called the *state-machine approach* [Sch93]. When a request arrives, it is received and processed by every replica, and every (correct) replica delivers a reply. The client waits until it receives the first reply (or a majority of identical replies if Byzantine conditions [LSP82] are present). This behavior is shown in Figure 3.1. Processing is shown in white in all figures.

This allows for quick failover if one of the nodes where a replica executes fails. The client issuing the request will then receive a reply from another node that did not fail, without much delay if any. This leads to low response time, even in the presence of failures. If there are  $k$  processes executing the replicas,  $k - 1$  of them can fail and the client will still receive a reply.

However, there are drawbacks associated with active replication. First, the redundant processing causes high resource usage. Second, and more

<sup>1</sup>The words “request” and “invocation” are used interchangeably from here on.

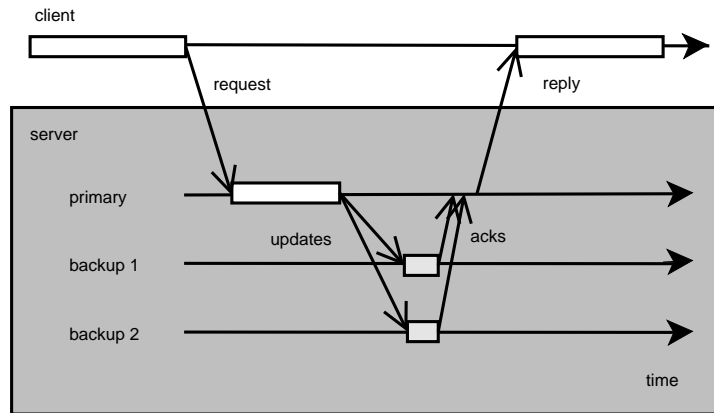


Figure 3.2: Passive replication

importantly, to be sure that each of the replicas has the same state-changes and act in a consistent way, thus satisfying linearizability, the execution must be *deterministic*. An operation is deterministic if the resulting output and state-change of it only depends on the initial state and the sequence of operations preceding the last one. Non-determinism can be caused by multi-threading, but other sources are also possible. [Pol93].

As the (deterministic) behavior of the replicas is determined by the sequence of operations already performed, it is required that all the replicas receive the operations in the same order. Atomic multicast (discussed in Section 3.2.1) is required to ensure this.

### 3.1.2 Passive Replication

In passive replication only one of the replicas receives, processes and replies to a request. This replica is called the primary, and the rest are called backups. Because of this, passive replication is also called the *primary-backup approach* [BMST93]. As shown in Figure 3.2, the primary sends update messages (state-changes or checkpoints) to the backups when it has finished the processing. When a backup receives an update it applies it and sends an acknowledge *ack* message back to the primary. When the primary has received an *ack* from every backup, the reply to the client is sent. Updating is shown in grey in all figures.

If the primary fails, the backups use a group membership service (see Section 3.2.2) to elect a new primary. The new primary performs recovery from the last checkpoint and then continues the processing of requests. If a backup fails, the server group excludes it from the group protocol, while the primary continues to process requests.

As a failure of the primary will increase the response time, it might not be suitable for real time system with rigid response time requirements.

However, since only the primary performs the processing, this approach does not entail a high resource usage and the execution can be non-deterministic.

A variant of passive replication, coordination cohort [BJRA85], specifies that client requests are sent to all replicas, not just the primary. Another variation, semi-passive replication (see Section 3.1.4), does the same.

The approach described here clearly satisfies linearizability in the failure-free case. However, if the primary fails, the request from the client might be lost. This requires the client to re-issue the request (driven by a timeout mechanism) and thus the failure is not masked. This is easily fixed if the client assumes that the communication channel can lose messages. The problem is then reduced to a slower response time and discarding duplicate messages, both of which do not jeopardize linearizability.

### 3.1.3 Semi-Active Replication

This is a hybrid technique that allows both non-deterministic execution and active replication. One of the replicas is called the *leader* while the others are called *followers* [VBB<sup>+</sup>91]. As in active replication all the replicas process the request, but it is up to the leader to process the non-deterministic parts and notify the followers. The leader decides in what order requests should be handled and sends its decisions to the followers. Also, only the leader's reply is received by the client, because identical replies are removed by the communication system.

This technique has been developed in the context of Delta-4 [Pow91]. It is designed to be a dependable real-time architecture. Real-time systems require low response times, and needs to respond quickly to events. If, for instance, an alarm condition is raised, the event needs to be handled immediately. Thus, the specification also allows pre-emption of execution.

Linearizability is achieved because the leader controls which requests are executed, and group communication ensures that the order is preserved, thus satisfying the criteria.

### 3.1.4 Semi-Passive Replication

Semi-passive replication [DS02] [DSS98] is a variation of the passive replication technique. The client-server interaction, the timeout policy and the selection of the update values differs. As will be shown later, the latter is actually also a selection of the primary. Since only one of the replicas actually processes the request, non-determinism is allowed.

Figure 3.3 shows how semi-passive replication works in the failure free case. The client starts by sending the request to all replicas. The primary processes the request and proposes an update value to the backups. When a majority of the replicas have agreed upon the value, the primary sends the decision to all backups, updates its own state and replies to the client.

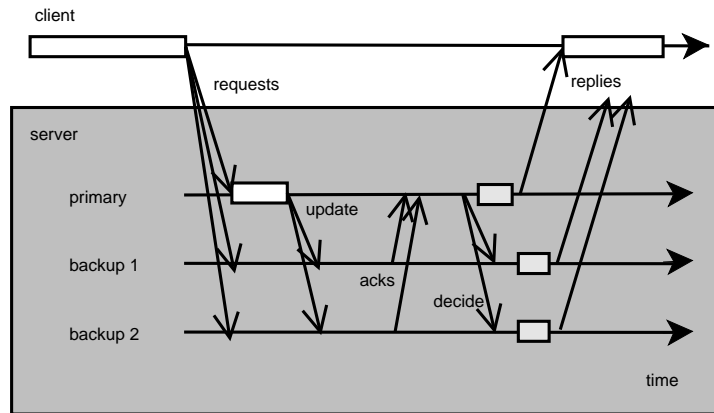


Figure 3.3: Semi-passive replication

As each of the backups receives the update, it updates its state and replies to the client. To mask the replication from the client, duplicate replies are discarded by the communication system.

If the primary fails or is suspected to have failed, one of the backups also propose a value. Since a process in an asynchronous system can be falsely suspected to have failed (see Section 3.2.2), two or more values might be proposed for the same request. A consensus protocol, presented in Section 3.2.3, solves this. (For details, see Section 3.2.3.) This has the effect of masking process failures in the server group from the client.

In standard passive replication, a process is removed from the server group if it is suspected to have failed. When it has recovered it may join the group again. Both of these operations, which are implemented in a group membership service (see Section 3.2.2), are time consuming and degrades the performance of the system. A timeout mechanism is usually used to detect a failed process. The length of the timeout is a tunable parameter. If an aggressive value is chosen, a slow process might be incorrectly removed from the group, and then subsequently rejoining it. This causes two invocations of the costly group membership service.

If a conservative value is chosen, a failure can lead to a long period where the system is just waiting for the timeout. For time-critical systems this is unacceptable.

Semi-passive replication avoids this problem by using two different timeout values; an aggressive timeout for the suspicion of a failure and a conservative for the exclusion of a process from the group. After the first timeout, but before the latter, the other processes keep sending messages to the suspected process. As only a majority of the processes needs to answer for the consensus protocol to reach an agreement, this will not affect the performance of the system.

## 3.2 Group Communication

All of the techniques presented in Section 3.1 require some form of internal communication between the members of the server group. Some examples are the following:

- Active replication requires a mechanism to ensure that a message is delivered in an atomic and ordered way.
- Passive replication needs to know which processes are members of the server group.
- Semi-passive replication needs to solve a consensus problem to agree on a result.

Remember that these are just examples of their use, and for instance, server group management is needed by all of the replication techniques.

Although some concepts will be briefly touched here, it is outside the scope of this report to go into all of the details. The parts that will be focused on here are those fundamental to the understanding of the techniques presented in Section 3.1 and the systems in Section 3.4. Interested readers are referred to [TS01, chapter 7] and [CDK01, chapter 11].

According to Baldoni et al. [BMT02], three systems that provide reliable group communication toolkits are ISIS [BR93], TOTEM [MMSA<sup>+</sup>96] and MAESTRO/ENSAMBLE [VB98].

### 3.2.1 Atomic Multicast

When a process wants to send a message to all of the members of a process group it would be (at best) a nuisance to send the message to one of the members at the time. Instead, a multicast service is used. To reliably deliver messages to a process group is a challenge. The message must be delivered to all or none of the processes. In addition, it is usually required that the order of the delivery must be the same for all processes as well.

To achieve this in an asynchronous system, where there are no upper bounds on message delay, *virtual synchrony* [TS01, pp. 387-389] is used. The message that is received by the communication layer will be delivered to the application only after it is safe to deliver it. All the processes in the process group must agree whether they deliver the message or not. The message is allowed to be discarded only if the sender crashes. This scheme allows for totally-ordered delivery and is called atomic multicasting.

### 3.2.2 Group Membership Service

A group membership service is needed to be able to create and delete groups, and must also support the joining and leaving of processes. To avoid a

SPOF, the service must be handled by a distributed protocol. A process can announce that it is joining the group, and if allowed, it must receive all messages sent to the group from that point. Similarly, if a process is leaving the group it must not receive any more messages [TS01, pp. 369-370].

So far this is pretty straightforward. But if a group member crashes it won't send out a message saying that it is leaving the group. Thus, some kind of mechanism for detecting process failures is needed. Unfortunately, to reliably and totally correctly detect such failures is hard. In fact, it can only be done in a synchronous system or a timed asynchronous system with a perfect failure detector. In a pure asynchronous system it is impossible to see the difference between a slow process and a failed one [FLP85]. Therefore, a process is only *suspected* to have failed.

A common way to implement this is using *heartbeat messages* [TS01, p. 715] in a *I-am-alive protocol*. All processes periodically send out a heartbeat message. If a process does not receive such messages from another process for a while, the process is suspected to have failed and removed from the group.

### 3.2.3 Group Consensus

The consensus problem is very similar to the problems of Byzantine generals [LSP82] and interactive consistency [CDK01, pp. 451-462], and they are all problems of *agreement*. The processes in the process group have to agree on a value out of a set of one or more values proposed by one or more processes in the group. They must agree even though processes may crash in the middle of the consensus protocol. As for the atomic multicast and the group membership problems this creates problems in asynchronous systems.

The *FLP impossibility result* states that in an asynchronous system “no consensus protocol is totally correct in spite of one fault” [FLP85]. It does not mean that it is impossible to ever reach an agreement, it merely means that there is a possibility that an agreement might not be reached. This can be circumvented by using a partially synchronous system [DLS88], restarting and recovering the failed process, use a probabilistic algorithm or make the system logically synchronous [CDK01, pp. 460-462].

Chandra and Toueg [CT96] shows that consensus can be reached in an asynchronous system even if unreliable failure detectors are used. It requires that less than half of the processes fail and that communication is reliable, and allows falsely suspected processes to continue their execution. This is the approach adopted in semi-passive replication (see Section 3.1.4, [DS02]). Chandra and Toeug [CT96] also show that consensus and atomic multicast (see Section 3.2.1) are reducible to each other. Thus, the solution is applicable to atomic broadcast as well.

### 3.3 Replicated Invocations

A *replicated invocation* happens when a replicated server invokes another server. The invoked server can be replicated, but it does not need to be. The resulting state-changes and result of such an invocation must be identical to an invocation not involving replicated servers. As will be evident soon, the problem to solve depends on whether the invoking replicated server is deterministic or not.

The two different problems will be presented here. Possible solutions to both are presented in Section 3.4.

#### 3.3.1 Deterministic Execution and the Duplicate Invocation Problem

Consider an actively replicated server  $A$  and a server  $B$ , that might or might not be replicated. Whenever  $A$  invokes an operation on  $B$ , each of  $A$ 's replicas sends a request to  $B$ . Since an actively replicated server must be deterministic (see Section 3.1.1), all of these requests are identical. If the request is idempotent, meaning that multiple invocations have the same end-effect as a single invocation, the problem is solved. However, since requests usually are non-idempotent, a mechanism to detect and handle duplicate requests is needed. This is called the *duplicate invocation problem*.

Figure 3.4 shows an example of the duplicate invocation problem for two actively replicated servers  $A$  and  $B$ . The client sends a request to each replica of server  $A$ ,  $a_0$  and  $a_1$ . They both process the request and issues a new request to the replicas of server  $B$ ,  $b_0$  and  $b_1$ .  $b_0$  and  $b_1$  will both receive two separate, but identical requests. The duplicated requests that must be handled are shown as light-grey arrows. As can be seen, there is also duplicate replies that must be taken care of. Fortunately, these can be handled by exactly the same mechanisms as the duplicate requests.

Passive, semi-passive and semi-active replication supports both deterministic and non-deterministic execution. If the server is deterministic, the problem is the same as for actively replicated servers. To see why, consider the following scenario: A primary replica sends a request to another server and then crashes. After one of the backups has become the new primary it must resend the last request. To the invoked server, this is a duplicate request. Although this example illustrates a passively replicated server, a similar argument can be made for semi-passive replication.

Semi-active replication can treat the communication with another server in two different ways. First, it could treat the creation of the request as a non-deterministic operation. This will make the invocations identical, giving the illusion that the execution is deterministic, thus facing the duplicate invocation problem. Second, it could treat the actual sending of the request as a non-deterministic operation and consequently only the leader will execute

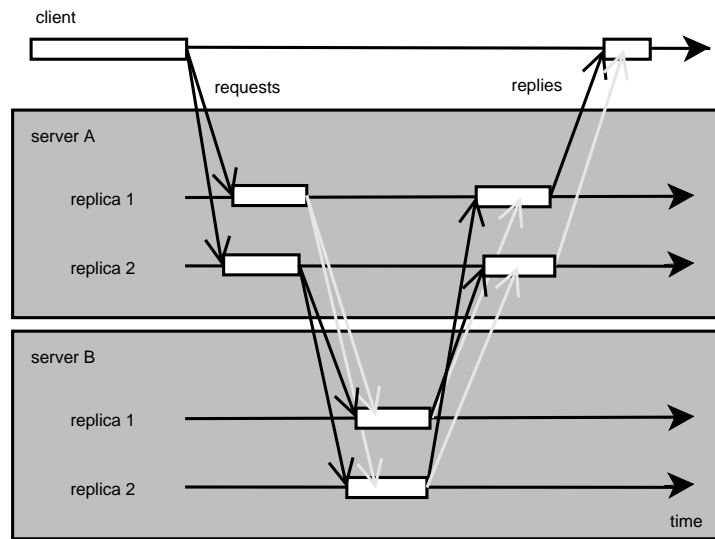


Figure 3.4: An example of a duplicate invocation

it. This will make the non-deterministic nature of semi-active replication opaque, as a failure of the leader might lead to a non-identical request being sent. This case is discussed in Section 3.3.2.

### 3.3.2 Non-Deterministic Execution and the Orphan Invocation Problem

A non-deterministic passively replicated server will normally behave as shown in Figure 3.5. The primary of server  $C$  receives a request, processes it and invoke a second server  $D$ . When a reply is received, the primary updates the backups and returns a reply.

Unfortunately, a serious problem appears if the primary crashes [PKS03b] [PKS03a]. To see this, consider Figure 3.6. If the primary of  $C$  sends a request to  $D$  and subsequently fails, one of the backups must become the primary. Because of the non-deterministic execution, there is no guarantee that the new primary will send an identical request to  $C$ . In fact, it might not invoke  $D$ , but a different server  $E$ , or it might not send the request at all.

If  $D$  processed the request received from the failed, old primary, it is now in an inconsistent state. And even worse; a subsequent invocation of  $D$  can lead to a reply that leaks the inconsistencies to the rest of the system. This is not acceptable. We call the request leading to the inconsistencies an orphan request, and the problem that needs to be solved is the *orphan invocation problem*.

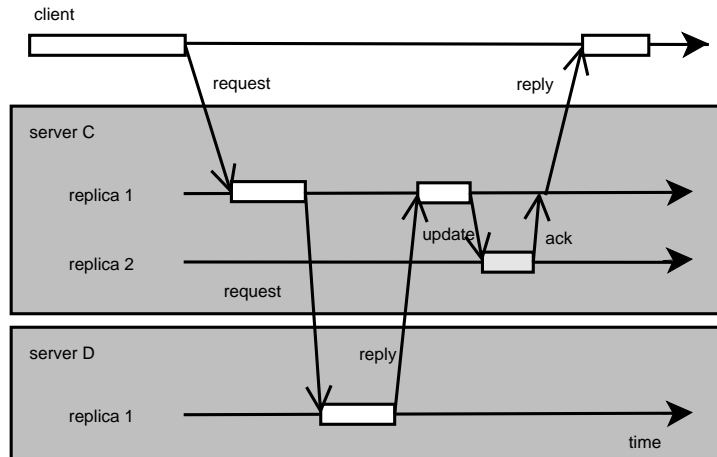


Figure 3.5: A non-deterministic passive server executing a replicated invocation

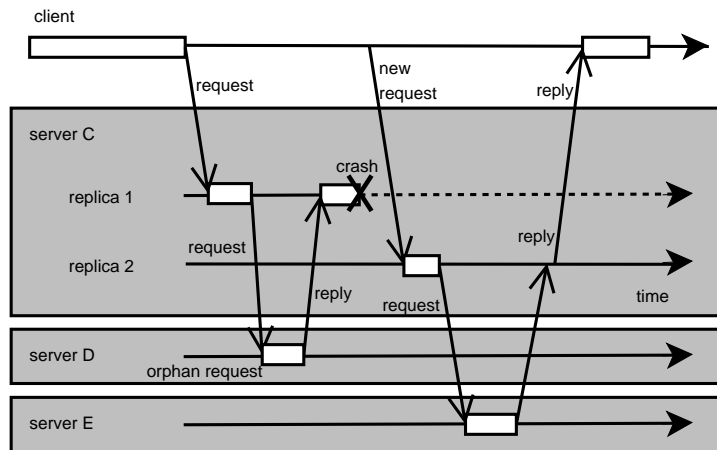


Figure 3.6: Non-deterministic passive replication and a primary failure lead to an orphan request

## 3.4 Systems Addressing the Replicated Invocation Problem

In the previous section, the seemingly inherent difficulties to replicated invocations were explained. Here, brief presentations of systems providing solutions for these problems are given. The two problems are considered in separate sections.

### 3.4.1 Deterministic Systems

Although it is quite straightforward to solve the duplicate invocation problem, the environments it is solved in varies. This section explains how some of the systems that solve the problem are designed.

First, an approach that unifies CORBA's OTS and FT-CORBA using a replicated transaction coordinator and gateways is presented. Second, ITRA and its client-side replication of the state are explained. Third, symmetric proxies and pre-filtering in GARF is described. Then, an approach for a massively replicated wide-area system is presented, and finally, a solution for a multi-threaded environment is described.

### Unification of Replication and Transaction Processing

Zhao et al. [ZMMS02] unifies two CORBA services; OTS and FT-CORBA. This approach is referred to as **Unification** in the rest of this document. In addition to replicating the application servers, it replicates the transaction coordinators. The latter leads to a non-blocking 2PC protocol, which is a highly desirable property because it ensures liveness.

The FT-CORBA standard defines in-bound gateways from the client to the replicated objects. To provide one-copy equivalence to the database servers, this approach advocates the need for out-bound gateways as well. Only the primary of the objects communicates with the gateway, and in the case of active replication one process is chosen for this purpose. For passive replication the primary is used. In this way, the database servers can be invoked by flat transactions through the XA interface [XA92]. These gateways are normally passively replicated to avoid single point of failures. If the primary gateway fails, this could lead to an abortion of all transactions currently being executed. To improve this, the gateways are also distributed. This means that if one of the distributed gateways fails, the only transactions that are aborted are the ones being processed by that particular gateway. By using both in-bound and out-bound gateways, the outside world can be indifferent of the replicated inner workings of the system.

Standard FT-CORBA is used for duplicate detection, suppression and scheduling between the replicas and objects. To support roll-back it is augmented to support the transaction related issues; the beginning and end,

abort or commit, of a transaction. Each user message is also associated with a transaction. This allows for automatic retry of aborted transaction without the knowledge of the client, which ensures exactly-once execution semantics as discussed in Part 1.

The FT-CORBA logging and checkpointing mechanisms are also improved to support stateful objects in a transactional environment. Normally, the checkpointing activity can take place when the object is operational quiescent. This is generally fine when the operations are independent, but in a transactional environment, they are not. Therefore, checkpointing can only happen when the object is transactional quiescent. This involves holding new transactions that are trying to access the object and wait until all current transaction have finished before a checkpoint is executed.

## ITRA

ITRA<sup>2</sup> [DG01] offers a passive replication scheme with recovery. The execution is made deterministic by making the result of a non-deterministic operation persistent over failures. This can be done by ensuring that enough of the replicas know about the result. This approach is unique in the sense that the state might also be replicated to the proxy. It also allows for reconfiguration of the system at run-time.

A proxy, sometimes called a stub, is the invokee's representative at the invoker. It is usually application independent. It allows the invoker to be indifferent of where and how the invokee is physically located. Instead, it just passes a request to the proxy of the invokee, which makes the invocation look local to the invoker.

In ITRA, each invocation is labeled with a unique and reproducible identifier. This allows a tier to filter duplicate messages at the invoked tier. Since this approach removes messages after they have been sent over the network, it is called post-filtering.

This approach is unique in the sense that it also allows replication of the server state to the proxy. In a system with little intra-tier replication, it improves the use of resources and the reliability of the system.

## Symmetric Proxies

Mazouni et al. [MGG95a] addresses the problem of *duplicated creations* in addition to duplicated invocations. Such creations occur when a replicated server creates a new object. The object is created by all replicas, thus making the system inconsistent. Luckily, the mechanisms to detect and remove duplicated invocations can also be used for duplicated creations.

---

<sup>2</sup>This approach was also presented as one of the systems in Part 1. More information about it can thus be found in Section 2.3.3.

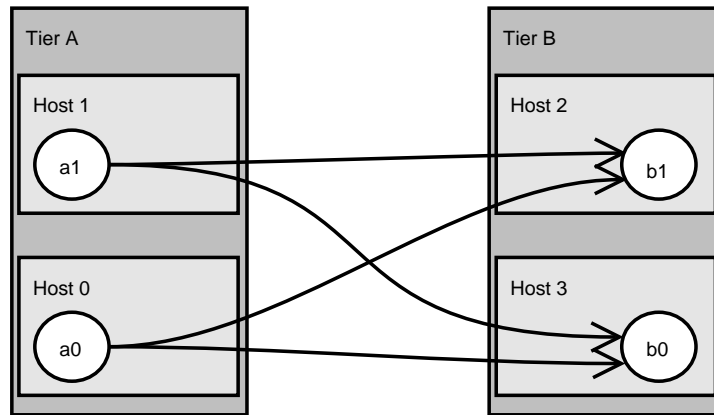


Figure 3.7: A duplicated invocation

Pre-filtering is introduced as a way of coping with the duplicate invocation problem in [MGG95a], while the role of the *symmetric proxies* are clearly presented in [MGG95b]. This work is done in an actively replicated system where one of the replicas is chosen as the primary in the sense that it is the only one that sends outgoing requests.

In a system using post-filtering and active replication, a replica in tier  $B$  will receive a request from each replica in the invoking tier  $A$ , as shown in Figure 3.7. The duplicate requests are removed by a mechanism at the invoked tier. If the message identifier has been seen before, the request is ignored.

Pre-filtering means suppressing the duplicate invocations at the invoker. The opposite is post-filtering, which imply the removal of duplicates at the invokee. Clearly, the first saves computer resources, especially network bandwidth, when compared to the latter. However, it is more complex in case of a failure. This will be shown later in this section. First, the role of the symmetric proxies is presented.

Proxies were explained in Section 3.4.1 in the context of ITRA. A symmetric proxy involves, not only an invokee proxy at the invoker, but also an invoker proxy at the invokee. This allows pre-filtering of both requests and replies as shown in Figure 3.8.  $a1$  is the primary at  $A$ , thus  $B$ 's proxy in  $a1$  sends a request to  $b1$  and  $b2$ .  $B$ 's proxy in  $a0$  knows that it is not the primary, and does not send. Therefore, each replica in  $B$  receives only one copy of the request.

In post-filtering, there is no need for failure handling if the servers are actively replicated. If  $a1$  fails, its invocation might never reach  $B$ , but unless all replicas of  $A$  fails  $B$  will at least receive one invocation. Pre-filtering, on the other hand, must handle failures. If  $a1$  fails, the surviving members of  $A$  elects a new primary. The new primary then invokes  $B$ . Since both the old and the new primaries' invocations might have reached  $B$ ,  $B$  must be

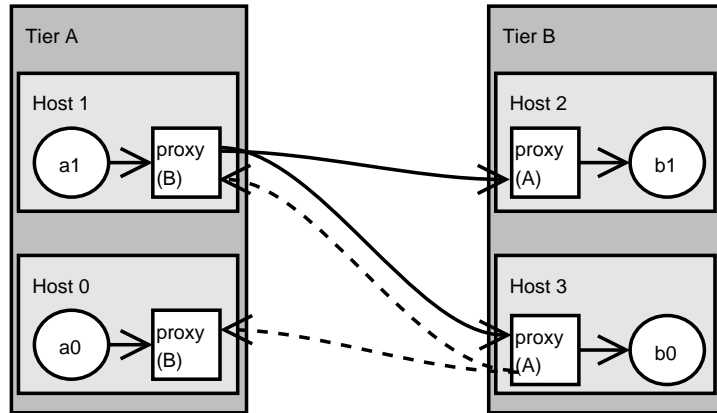


Figure 3.8: Pre-filtering in Symmetric Proxies

able to detect and discard duplicate messages. This is done in the same way as post-filtering. Therefore, in the presence of a failure, pre-filtering also requires post-filtering.

This approach, called **Symmetric Proxies** in this document, allows stateful servers as the state is kept persistent by replicating the servers and it was implemented in GARF [GGM94].

### Wide-Area Systems

In a wide-area system, group communication can not be assumed to be fast and efficient. A large number of replicas might make this even worse. It would be inefficient and slow to depend on the usual replication and group communication techniques.

Bakker et al. [BvST98] presents **Wide-Area Systems**, a solution for the duplicate invocation problem in such environments. The objects are actively replicated and the system distinguishes between read-only and modifying (write) operations. Read operations are only sent to a single and nearby replica, but write operations must be sent to all replicas. This is similar to the Read One Write All protocol presented by Helal et al. in [HBH96].

This protocol does not require a complete knowledge of all of the group members and uses nearby replicas if it can. Thus the load on replicas and communication system is reduced. The work assumes no network partitions and reliable communication channels.

Figure 3.9 presents the request phase of a write operation using this approach. *A*'s client-side proxy sends the request to all replicas of *A*. In addition to the usual request identifier there is a flag indicating whether the receiving replica of *A* is the primary or not. The flagged request is marked with a flag in Figure 3.9. *a1* receives the request marked with the flag and therefore proceeds to send a request to *B*. It also sends a hush

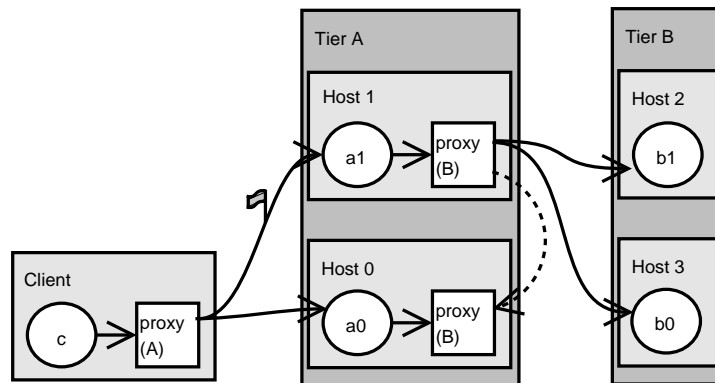


Figure 3.9: A write request using hush-messages in Wide-Area Systems

message (shown as a dotted arrow) to all other (known) proxies of  $B$  to save network bandwidth. The hush message includes the request identifier and the previous reply of a write request  $a1$  received. The reason for this is that the state might not be consistent in all replicas because of the lacking group communication. In particular, linearization is jeopardized.

In the reply phase each replica in  $B$  sends the reply to a subset of  $A$ 's replicas, which returns the reply to the client.

A read request is only sent to one replica. Since the linearization criterion might not hold, the read request must check if the version state of the replica is the correct one. If not, a replica containing the correct state must be found.

If a proxy does not receive a reply it resubmits the request to provide exactly-once semantics to the client. Since duplicate invocations are post-filtered, there is no danger in resubmitting requests, even to other clients.

This approach only supports one client and needs to be extended to a multi-client environment.

### Circus

A *troupe* is defined by Cooper in [Coo85] as a set of replicas of a server. It is actively replicated and each replica is deterministic. However, servers in **Circus** [Coo85] might be multi-threaded, thus complicating the synchronization of the troupe.

Post-filtering is used since a request is sent by all client replicas to all server replicas. In base **Circus**, servers wait for a request from each available client before it continues, and clients wait for a reply from each available server before it returns. A client will receive a notification if the server replica has crashed. Thus, the return of a reply is a synchronization point where all server and client replicas are respectively in the same state. This is sufficient in a single-threaded environment, but a more elaborate serialization concept is required in a multi-threaded system.

The definition of a correct transaction is that it is *serializable* [BHG86]. Serializability means that the result of executing multiple transactions in parallel is the same as some serial execution of the same transactions. If the troupe is multi-threaded the system needs to make sure that transactions are serialized in the same order on *all* replicas. This can be done by any kind of concurrency control algorithms [BHG86].

The Optimistic Troupe Commit Protocol (OTCP) is used in `Circus`. It is optimistic because it assumes few conflicts. It does not pose any restriction on the local concurrency control mechanisms used by the replicas and it does not require any communication between the members of the troupe. An attempt to serialize transactions differently is detected by the protocol and transformed into a deadlock. Thus, `Circus` executes transactions correctly in a multi-threaded environment.

### 3.4.2 Non-Deterministic Systems

As discussed in Section 3.3.2 a server using passive replication and supporting non-deterministic execution can cause orphan requests. In the case of nested transactions, the orphan request is actually a subtransaction.

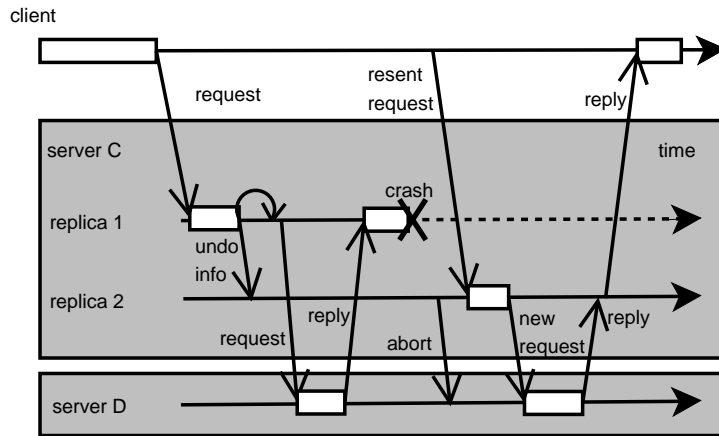
#### Preventing Orphans

Pleisch et al. [PKS03a, PKS03b] describes two schemes to solve this problem, an optimistic and a pessimistic. An optimistic server commits a (sub-) transaction as soon as it returns a reply. If the parent process aborts, a compensating transaction is needed to undo the effects of the committed transaction. This approach promotes liveness as it does not block any part of the server while the parent transaction is still executing, but jeopardizes safety as another transaction may read the result of a committed transaction that might later be compensated.

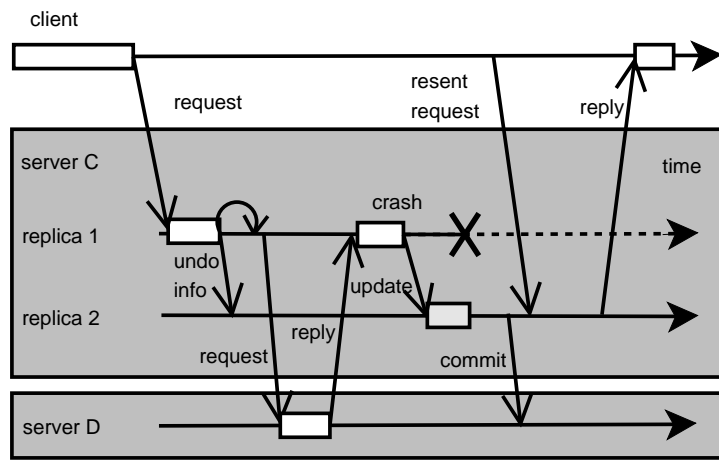
A pessimistic server does not commit a transaction until the parent transaction is committed. The locks that are held by the transaction block other transactions that might be trying to access the same objects. If the parent transaction crashes the locks are held indefinitely. This approach promotes safety as no inconsistencies can arise, but jeopardizes liveness since transactions can be blocked forever. Thus, a mechanism to remove such locks is needed.

Summing-up, the following is required to fix the problem:

- The termination of unterminated orphan subtransactions on pessimistic servers, and
- the compensation of terminated orphan subtransactions on optimistic servers.

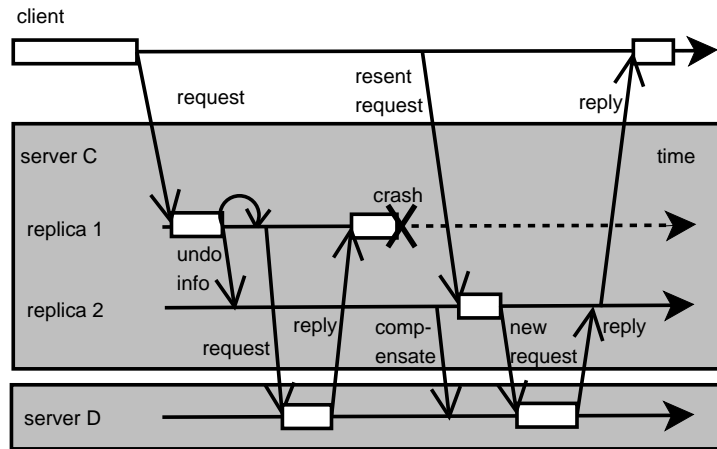


(a) Failure before updating backups

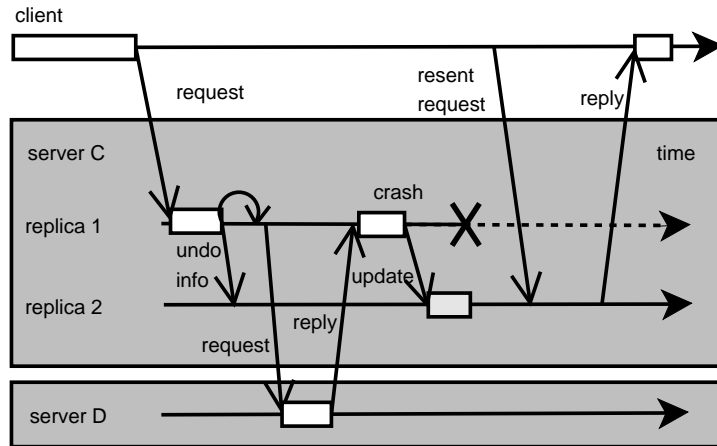


(b) Failure after updating backups

Figure 3.10: Passively replicated non-deterministic server *C* invoking pessimistic server *D* in Preventing Orphans



(a) Failure before updating backups



(b) Failure after updating backups

Figure 3.11: Passively replicated non-deterministic server *C* invoking optimistic server *D* Preventing Orphans

The solution proposed, called **Preventing Orphans**, for both optimistic and pessimistic servers, is to send undo information from the primary to the backups before the invocation. For a pessimistic server this undo information holds information that enables the new primary to terminate (commit or abort) the old primary's request. For an optimistic server the information contains enough to enable the new primary to locate and compensate the previously committed subtransaction. Note that this compensation is anything but straightforward, since the execution sequence  $(request_x; request_y; compensate_x)$  must leave the server in a consistent state [BHG86].

As can be seen in Figure 3.10 and 3.11, the primary of server *C* sends undo information to the backups before invoking *D*. If the primary fails *before* updating the backups, as shown in Figures 3.10a and 3.11a, the new

primary must respectively abort or compensate the subtransaction, before it can continue serving the client. The client will most likely resend the same request driven by a timeout. It will be processed by the new primary as new request.

If the primary fails *after* updating the backups, two other scenarios take place. If  $C$  invoked a pessimistic server (see Figure 3.10b), the new primary must commit the subtransaction. If the invoked server is optimistic (see Figure 3.11b), nothing needs to be done. The subtransaction is already committed. In both cases the client resends the request, and the previously calculated result is returned to it.

This approach avoids a centralized transaction coordinator and allows stateful replicas. Since no recovery is discussed, crashed replicas are assumed not to recover to ensure a non-blocking scheme.

### Reconciling Replication and Transactions

Felber and Narasimhan [FN02] proposes, in much the same way as **Unification** by Zhao et al. [ZMMS02] (see Section 3.4.1), a reconciliation of OTS and FT-CORBA to provide both safety and liveness for CORBA. The most important difference is that this approach extends FT-CORBA to allow non-deterministic execution of replicas. Naturally, this prohibits active replication. This scheme is referred to as **Reconciling**.

As in **Unification**, the client needs only to conform to the client-side specification in FT-CORBA standard; being able to iterate to the profiles in the IOGR and resend invocations. Also, each request is associated with an OTS transaction. This might be a nested transaction if it is a replicated invocation.

Felber and Narasimhan [FN02] presents an optimistic protocol. It is optimistic in the sense that it commits subtransactions before the parent transaction. The essence of the protocol is shown in Figure 3.12. The client invokes server  $C$ , which again invokes server  $D$ . The primary replica of  $D$ ,  $d1$ , processes the request, updates the backups and sends a reply to  $C$ . Meanwhile,  $c1$  has crashed and the reply never reaches its destination. Thus,  $d1$  must roll-back the transaction it just committed.

The client-side FT-ORB detects the failure of  $c1$  and transparently invokes  $c2$ .  $c2$  treats this as a new request and invokes  $D$ .  $d1$  processes the request, updates the backups and commits the subtransaction. Then, it crashes in such a way that the reply is not received by  $c2$ . The FT ORB at  $c2$  notices the failure of  $d1$  and sends an identical request to  $d2$ . Since  $d2$  has been updated, it just fetches the result, commits the subtransaction and replies to the invoker.  $d2$  can then complete its processing, commit the transaction and return a reply to the client.

Note that  $d1$  notices that  $c1$  has crashed. It is actually the FT ORB at  $d1$  that is notified of the failure. This enables  $d1$  to rollback without being

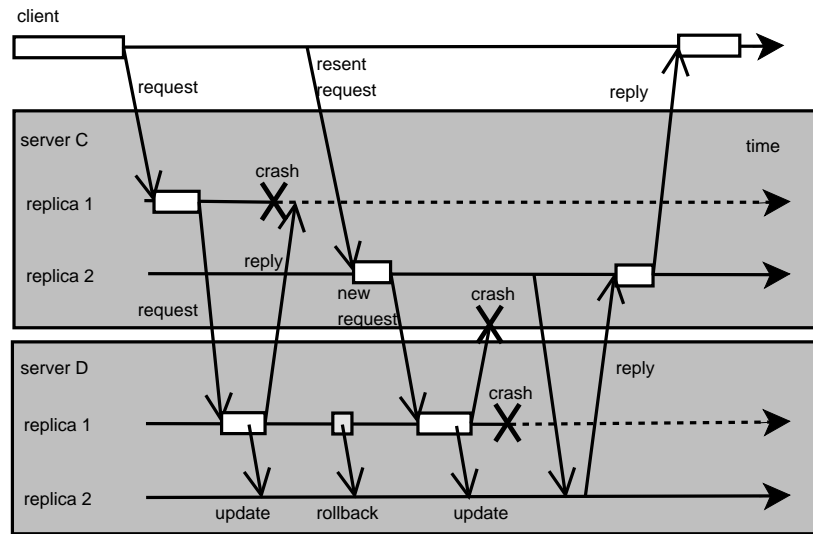


Figure 3.12: Two failures at two different and passively replicated servers in Reconciling

explicitly rolled back by the invoking tier *C*. Contrast this with the approach by Pleisch et al. in [PKS03b], presented in Section 3.4.2. It assumes that this detection is not possible, and thus undo information is needed. By being able to detect the crash at the invoking tier, the need for sending the undo information to the backups is circumvented.

## Chapter 4

# Analysis and Discussion

As the systems presented in Chapter 2 and 3 of this report have been designed for various applications and different choices have been made during the development process, their approaches and solutions differ. In this chapter, the most important and influential properties of Chapters 2 and 3 are compared and analyzed.

Sections 4.1 and 4.2 compares and discusses the systems in Chapters 2 and 3, respectively. Then, a need for combining the concepts and an example of how it can be done is presented in Section 4.3.

### 4.1 Exactly-once and Transactions

The systems presented in Chapter 2 all have their various ways to deal with failures. Most of them offered exactly-once execution semantics, stateful middle-tier servers and some kind of recovery. About half of them considered the interaction with the end-user, and some mentioned or used replication. A short summary of this is given in Table 4.1. The following subsections summarize each of the properties separately.

#### 4.1.1 Stateful Tiers

If application servers are allowed to keep their own state, it must be kept persistent over failures. Because stateless programming is, at best, inconvenient, stateful middle-tier servers simplify application programming. **Direct TP** and **Queued TP** only allow state(s) in the queues and end-tier databases between transactions. As **TEO** is, in principle, just a normal TP-system, it too only permits state in the databases. **iSAGA**, and of course the two-tier approach in **Input and Output Lists**, store the entire state in the database.

**Phoenix/APP** uses message and state logging to make the state persistent for all Pcoms, and **ITRA** uses replication of the state both within the tier

	Stateful Tiers	Execution Semantics	End-User Interaction Control	Replication	Recovery
Direct TP	End-tier	At-most-once	No	No	Undo-Redo
Queued TP	End-tier and queues	Exactly-once	Yes	No	Undo-Redo
TEO	End-tier	Exactly-once	No	Passive	No
Phoenix/ APP	End- and middle-tiers	Exactly-once	Yes	No	Redo
ITRA	End- and middle-tiers, plus proxy	Exactly-once	No	Passive	Redo
CORBA	End and middle-tiers	Exactly-once	No	Passive and active	Undo-Redo
iSAGA	End-tier	At-most-once	Partial	No	Redo
Zero-Loss	Middle-tier	Exactly-once	No	Passive	No
Input and Output Lists	End-tier	Exactly-once	Yes	No	Undo-Redo

Table 4.1: A comparison of the systems in Chapter 2

and to the proxy in the invoking tier. **Zero-Loss** has no description of database tiers, and consequently state is kept in the replicated application servers. Although the **CORBA** standard supports stateful applications, all known implementations are stateless.

### 4.1.2 Execution Semantics

As explained in the introduction in Section 2.3, **Direct TP** provides only atomicity and thus at-most-once execution semantics. There are several ways to strengthen this exactly-once semantics. Persistent queues, message logging, automatic retry of failed transactions, involving the client in the commit decision and using object groups are presented in Chapter 2. As **iSAGA** does not use any of these techniques, it can only provide at-most-once.

**Phoenix/APP** and **ITRA** logs requests and replies, **Queued TP** uses queues and **TEO** resends requests and restarts aborted transactions. **CORBA** provides object groups and restart of transactions, while **Zero-Loss** uses the dispatcher to provide exactly-once. The atomicity of removing a request from the input list to logging the value, and likewise for writing the output on the terminal and appending it to the output list, provides exactly-once for **Input and Output Lists**.

### 4.1.3 End-User Interaction

As discussed in Section 2.4, the interaction with the end-user must be controlled to be able to mask failures in the best possible manner. The section concluded that, a *complete* masking is not only unrealistic, but impossible; There is always a window of opportunity for a non-masked failure, but it should be made as small as possible.

**Direct TP**, **ITRA**, **TEO**, **CORBA** and **Zero-Loss** does not handle end-user interaction at all.

**Queued TP** and particularly the approach by Bernstein et al. [BHM90] employs queues to make the interaction as controlled as possible. **Input and Output Lists** approach is very similar to the queues, while **Phoenix/APP** captures the interaction in the eXternal Interaction Contract.

**iSAGA** has no explicit handling of end-user interaction, but presents the user with its last acceptable view of the system state after a failure.

### 4.1.4 Replication

Replication is an important technique for achieving fault-tolerance and masking failures, and it is thoroughly presented in Chapter 3. A short summary of the systems in Chapter 2 is given here.

**Direct TP**, **Queued TP**, **iSAGA** and **Input and Output Lists** do not handle replication. **Phoenix/APP** states that servers can be replicated for performance or fault tolerance, but no details are given.

**TEO**, **ITRA** and **Zero-Loss** uses passive replication and the **CORBA** standard supports both passive and active replication.

#### 4.1.5 Recovery

Sooner or later a server is bound to fail. Unless there are an infinite number of servers available, the system must be able to recover a failed server. A quick recovery of a failed node will increase the availability of the system. Since a thorough study of recovery is beyond the scope of this report, only a short summary will be given.

All systems that only have state in databases or in queues can use standard undo and redo recovery to recover from failures. Thus, **Direct TP**, **Queued TP** and the database in **Input and Output Lists** can easily rely on these mechanisms. **CORBA** utilize the Object Transaction Service, while **Phoenix/APP** and **ITRA** relies only on redo for application servers as requests and replies are logged. **Zero-Loss** and **TEO** does not support recovery at all, and **iSAGA** recovers to the last valid view for both the database and the user.

## 4.2 Replication and Replicated Invocation

The key properties of the systems presented in Chapter 3 are compared in this section. These are: Type of replication, deterministic execution or not, the support for transactions and replicated invocation handling. The systems are compared for these properties in the following subsections and a short summary is given in Table 4.2.

### 4.2.1 Replication

Replication is crucial to achieving high availability. None of the systems discussed here used semi-active or semi-passive replication. **Unification**, **Symmetric Proxies** and **Circus** use active replication, while **Wide-Area System** has variant of active replication where writes are processed by all replicas and reads are processed by only one replica. **ITRA**, **Preventing Orphans** and **Reconciling** are passively replicated schemes.

### 4.2.2 Execution

Whenever a server is actively replicated it must be deterministic or rendered deterministic like semi-active replication does. Consequently, **Unification**, **Symmetric Proxies**, **Circus** and **Wide-Area Systems** are deterministic systems. **Circus** also offers a multi-threaded option for the servers. This raises

	Replication	Execution	Transaction Support	Replicated Invocations
<b>Unification</b>	Active (Passive gateways)	Deterministic	Yes	Post-filtering and gateways
<b>ITRA</b>	Passive	(Non-) Deterministic	Yes	Post-filtering
<b>Symmetric Proxies</b>	Active	Deterministic	No	Pre-filtering
<b>Wide-Area Systems</b>	Active (Read One Write All)	Deterministic	No	Pre-filtering
<b>Circus</b>	Active	Deterministic and multi-threaded	Yes	Post-filtering
<b>Preventing Orphans</b>	Passive	Non-deterministic	Yes	Backup undo information
<b>Reconciling</b>	Passive	Non-deterministic	Yes	Post-filtering

Table 4.2: A comparison of the systems in Chapter 3

some issues of non-determinism that is handled by the OTCP. Although **ITRA** allows non-deterministic events, these events are handled in the same way as non-deterministic events in semi-active replication. Thus, the execution of each tier looks deterministic for all other tiers. **Preventing Orphans** and **Reconciling** allows non-deterministic execution.

### 4.2.3 Transaction Support

As demonstrated in Chapter 2, the atomicity of transactions is important to achieve exactly-once execution semantics. Therefore, highly available systems should provide support for transactions.

**Unification** uses OTS and replicates the transaction coordinator to achieve a non-blocking transaction service. **Reconciling** also uses OTS to handle transactions.

**ITRA** is suggested by the authors to give better support for long transactions and transactions with non-rollbackable (real) operations by combining its unique scheme with a replicated transaction manager. **Circus** uses a distributed protocol (OTCP) for concurrency control of concurrent transactions.

**Preventing Orphans** provides a distributed commit protocol to avoid a centralized blocking transaction coordinator.

Neither **Symmetric Proxies** nor **Wide-Area Systems** mention anything about support for transactions.

#### 4.2.4 Replicated Invocations

As explained in Section 3.3 replicated invocations need to be controlled. **Circus** and **ITRA** uses duplicate identification at the invokee to filter out the replicated invocations. This is also known as post-filtering. **Unification** and **Reconciling** is as the CORBA standard also based on post-filtering, but the first adds gateways to prevent duplicate request from reaching the database tier. The latter allows an orphan request to be notified and rolled back if the parent process crashes.

**Symmetric Proxies** and **Wide-Area Systems** use pre-filtering. Both designate a primary that is responsible for sending outgoing messages. The latter approach chooses the primary for each tier on a request-by-request basis, while the first is more permanently chosen. The latter also sends hush messages to the others replicas.

**Preventing Orphans** sends undo information to the backups to enable them to terminate or rollback the orphan request if the primary fails.

### 4.3 Combining Forces to Achieve Availability

As seen in Chapters 2 and 3, a lot of different strategies and techniques are used for different purposes to achieve various goals. The main goal is availability by failover and failure masking. Each of the systems presented gives a unique solution or approach to the problem. A complete fault tolerant system must provide availability of the services offered, keep the state of the system consistent, and be able to mask and recover from failures. To fully achieve this, a need for combining the strengths of the key mechanisms arises.

The key mechanisms that are required are:

- *Transactions* provide the safety property. They are needed to clean up after failures and to keep the system consistent, without using messy ad-hoc techniques [GR93]. It is also an important part of exactly-once execution semantics as it provides at-most-once.
- *Replication* is needed to provide fault-tolerance and the liveness property. It makes it possible to avoid long outages and very slow responses to clients.
- *Exactly-once* execution semantics is needed to mask all failures. Explicit *end-user interaction control* provides a way for the end-user to determine the outcome of the last requests made, even if the client should fail.
- *Recovery* is needed to bring failed processes back online in a consistent state. Without recovery, an infinite number of replicas are needed, since it is just a matter of time before all replicas have failed, and the system is rendered unavailable.

Transactions, exactly-once and replication have been discussed in this report. The last mechanism, recovery, has not been presented thoroughly here because of space limitations. Good introductions can be found in [TS01], [CDK01] and [GR93]. By assuming an infinite number of replicas, the need for recovery can be alleviated. Remember though, that in a real high availability system, recovery is also required.

Fault tolerance is needed both for protecting the data and for protecting the processing. Transactions are often used as a way to protect data, while replication is used for the protection of processing. Together, these two provide availability and consistency for a system. If these two mechanisms are integrated rather than being kept as separate concepts, two additional properties can be provided [FN02]:

- A “stronger consistency to replicated systems by supporting non-deterministic operation”, and
- a “higher availability and failure transparency to transactional systems”.

The former allows stronger consistency and non-deterministic execution because orphan requests can be rolled back in transactional systems, while the latter provides higher availability and failure masking because replication allows the execution to be rolled forward to another replica. If exactly-once execution semantics with explicit end-user control is added on top of these, an available, consistent and failure masking system that supports non-deterministic execution emerges.

First, Section 4.3.1 describes in details the two general failures scenarios between a client and a user that are impossible to mask, and a way to compensate for them. Second, the integration of the end-user interaction control and the transaction concept is presented in Section 4.3.2. Then, Section 4.3.3 offers some general guidelines for how to combine replication, transactions and exactly-once with end-user interaction control, and which approaches are best to integrate. Finally, an unification of **Preventing Orphans** and **Phoenix/APP** is presented in Section 4.3.4.

### 4.3.1 Unmasked Failure Scenarios

As mentioned in Section 2.4, there are two failure scenarios that are impossible to mask from the user. They both involve a client crash. The first scenario takes place when the user has sent a request to the client, but the client crashes before the request is made persistent. The only way to remedy this is to decrease the window of opportunity, in which such a failure can occur, as much as possible. This is done by making the request persistent as soon as possible. There are two ways to ensure the persistence of a request. Either replicate it to one or more other nodes with independent failure

modes, or force it to persistent storage at the local node. Using today's technology, a disk write is generally slower than a communication session, thus the latter is preferably used, even though it generates more network traffic.

The second failure scenario takes place if the client has sent a reply to the user and crashes before it is sure that the user has seen the reply. When the client is recovered it must decide whether it should resend the reply to the user or not. If it resends the reply, it might be seen multiple times by the user, but if it does not, it might not be seen at all. Generally, it is a better idea to show the user multiple outputs instead of none. At least that way, the user does not assume that the request has been lost and sends a new one.

To compensate for these failure scenarios, the user should be able to get an answer to the question: What was my latest logged input to the client, and what was the corresponding output? It can easily be answered. As the final step of the client recovery, the last input is fetched from the log and shown to the user. If the corresponding reply is also in the log, it is also shown. If not, the request is resent and when the reply is received it is shown as standard output.

### 4.3.2 Consistent Output from Transactions

The two failure scenarios are general problems of end-user interaction. In a transactional system another issue occurs: The end-user interaction must be integrated with the transaction concept. This must be done in such a way that the outcome of a transaction will not be accessible to the user until the transaction is sure to commit. If a transaction produces a reply that is shown to the user before it commits, the user may act on the reply, issuing a second transaction versus the system. If the original transaction later aborts, the second transaction might have been started on a false foundation. Although transactions like the last one do not directly violate the consistency of the system, it does so indirectly, because the user interacting with it makes a decision on an uncommitted and hence inconsistent (since the transaction is rolled back), state of the system.

Real actions are operations that a system is performing against the real, physical world, and once done, they cannot be undone [GR93, pp. 164-165]. By treating the output to the client as a real action, it is delayed until the transaction is certain to commit. Only then can the consistency of the reply be guaranteed and the client is free to display the result to the user. Precisely how this reply is delivered to the user depends upon whether the client is thin or thick, and which mechanism ensures exactly-once execution semantics in the given system. Of the approaches described in this document, `Phoenix/APP` uses message logging, `Queued TP` uses queues and `Input and Output Lists` uses lists.

### 4.3.3 Combining Approaches

The author has not found a single approach with support for all of the key mechanism discussed in this report; transactions, replication and exactly-once including end-user interaction control:

- Replication and TP is merged in **TEO**, **ITRA**, **Circus** and **Preventing Orphans**, and although **CORBA** has support for both, they were developed as independent services, and an integration like the one presented in **Reconciling** or **Unification** is needed. None of these have explicit control of the end-user interaction.
- The three systems providing exactly-once with end-user interaction control are **Queued TP**, **Phoenix/APP** and **Input and Output Lists**, but none of these has support for replication.

Thus, the techniques presented in different sections of this report should be combined to support all of these key mechanisms.

It seems to be easier to begin with a concept with merged support for replication and transactions and extend it to support end-user control, than the other way round. This is backed up by the observations that the end-user interaction control is a smaller and easier part of a system, and that exactly-once is usually already implicitly achieved in a replicated system by its replicated nature. Thus, only the interaction between the client and the end-user needs to be modeled. The systems that seems to be most promising and best suited to be extended is **ITRA**, **Circus**, **Preventing Orphans** and the **CORBA** based systems; **Reconciling** and **Unification**. An important difference of these approaches is that **Circus** is actively replicated, **ITRA**, **TEO** and **Preventing Orphans** is passively replicated, and the **CORBA** based systems are both.

**Queued TP** requires the overhead of two extra 2PC decisions to guarantee exactly-once, and **Input and output Lists** needs to make the removal of elements from the input-list and the insertions into the output-list atomic. As they both involves extra resource usage compared to **Phoenix/APP**, the latter is chosen as the way to control the end-user interactions. This requires a thick client.

**Preventing Orphans** has integrated support for both TP and replication. It is not restricted to deterministic processing (like **Circus**) or stateless application servers (like **TEO**), and does not involve a centralized commit coordinator (like **Reconciling** and **Unification**). Unlike **ITRA**, where exactly how the integration of transactions is done is unclear, **Preventing Orphans'** transactional support is thoroughly described in [PKS03a, PKS03b]. Thus, it is chosen as the preferred approach to be extended with the end-user interaction control of **Phoenix/APP**. This extension is showed in the following section.

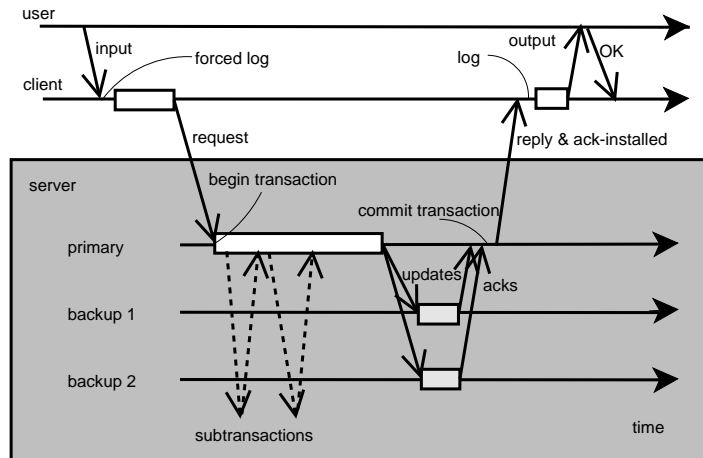


Figure 4.1: A failure free run of the combination of Phoenix/APP and Preventing Orphans.

#### 4.3.4 A Combination of Preventing Orphans and Phoenix/APP

Whenever a transactional system is extended to support exactly-once execution semantics with control of the end-user interaction, the reply must be treated as a real action as previously described. To facilitate this, the mechanism that controls the end-user interaction must be integrated with the protocol that ensures the atomic commit or abort of the transactions, typically 2PC. Also, when a new request is initiated by the user, a new transaction must be started. As **Preventing Orphans** has no explicit description of duplicate detection handling, a **Phoenix/APP** CIC is used for the request interaction between a client and a server.

In **Phoenix/APP**, input from the user is forced to persistent storage at once it is received by the client, as a part of XIC. This minimizes the window of opportunity for a failure that can cause the input to be lost. The integration with the transactions in **Preventing Orphans** is as shown in Figure 4.1. The request is sent from the client to a replicated server. A new transaction is started by the primary of the server, and the request is associated with that transaction. The transaction might initiate distributed nested subtransactions, each of which will cause undo information to be sent from the primary to the backups. The undo information is not shown in the figure to avoid cluttering it. When the transaction has finished the processing, it updates the backups, commits and returns a reply and an *ack – installed* to the client. An *ack – installed* tells the client that it does not have to resend the request ever again (see page 12). The client logs the reply before it is shown to the user. At a later point, the user acknowledges that it has seen the reply i.e. by pressing an “OK” button. The client can then be sure that

the user has seen the reply, and that the invocation is completed.

Notice that because the server is replicated, the primary does not need to log any of the interactions to guarantee its persistence. It is simply enough to replicate the information to the backups. This is done as a part of the update.

In this combined approach the following client and server failure scenarios can occur:

- *C1*: The client fails before the input has been forced to the log.
- *C2*: The client fails after the input has been forced to the log, but before a request is sent to the server.
- *C3*: The client fails after the request has been sent to the server, but before the log entry of the reply is persistent.
- *C4*: The client fails after the log entry of the reply is persistent, but before it knows whether the user has seen the output.
- *C5*: The client fails after it knows that the user has seen the output.
- *S1*: The primary fails before the backups have been updated.
- *S2*: The primary fails after the backups have been updated.

For scenario *C1*, the failure is not masked and the user should re-enter the input. For *C2*, the recovery protocol of the client will find the input in the log, but no corresponding log entry for a received reply. Thus, the client sends the request. This is the same for scenario *C3*, but here the request is a duplicate, and it is up to the duplicate detection mechanism in the server to decide what to do: If the server is still executing the corresponding transaction it is discarded, and if a reply has already been sent by the server, it is re-transmitted. After a *C4*, both the request and the reply are found in the log and there is no reason to resend the request. Because an output might have been shown to the user before the crash as well, cases *C3* and *C4* can cause the client to see duplicate replies. The last client failure scenario, *C5*, is trivial since there are no unfinished requests or replies to process.

After an *S1*, the server forgets all about the request. However, as a part of a client side CIC between the client and the server, it is periodically resent by the client. Thus, the new primary will receive the request and begin a new transaction. For scenario *S2*, the new primary does not know if the client has received the reply, but it is assumed to have received it. If the client has not received a reply, the client will, as for *S1*, periodically resend the request. Upon receiving a duplicate request the server replies with the original reply.

By combining the end-user interaction control of `Phoenix/APP` and the transaction and replication mechanisms of `Preventing Orphans`, the system

is highly available and consistent and all failures are masked from the user. Exceptions are *C1* and possibly *C3* and *C4*. But, as discussed earlier, these failures cannot be completely masked from the user.

## Chapter 5

# Conclusions and Further Work

The ultimate goal in fault-tolerant computing is to provide a continuously available system. Sadly, as stated by Murphy's Law, it is always going to be a small probability that a system will be unavailable at some point in time. Therefore, such a system is not feasible. The only thing we can do is to make a best effort to get as close as possible to that goal.

As discussed in Chapter 4, elements of replication, transactions, exactly-once execution semantics and recovery should be incorporated into a single, highly available system that guarantees end-to-end failure masking. Some of the systems presented have support for all four of these mechanisms, and others have integrated two or three mechanisms, but none of the systems have a complete integration of all of them. Although, a short outline for how an integration of the first three of these concepts can be done has been given, a detailed description of a system that integrates all of them in a seamless way is still needed. How to design and implement such a system is clearly a possible issue for future work.

A closer look at the end-user interaction, shows that none of the approaches presented here consider the possibility of showing the user its last logged input to the client as the final part of recovery. This allows the user to see whether the client received its last input before the failure. Although this will not cause the failure to be masked, it is mitigated. Also, if a client fails and cannot be restarted at once, a user might move to another client. How to handle this scenario is not discussed by any of the approaches in this report.



# Bibliography

- [AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *j-IPL*, 21:181–185, 1985.
- [BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.
- [BHM90] Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing recoverable requests using queues. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 112–122. ACM Press, 1990.
- [BJRA85] Ken Birman, T. Joseph, T. Raeuchle, and A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, June 1985.
- [BLBA00] Roger S. Barga, David B. Lomet, Thomas Baby, and Sanjay Agrawal. Persistent client-server database sessions. In *EDBT Conference*, pages 462–477, March 2000.
- [BLP<sup>+</sup>03] Roger Barga, David Lomet, Stelios Paparizos, Haifeng Yu, and Sirish Chandrasekaran. Persistent applications via automatic recovery. In *IDEAS Conference*, July 2003.
- [BLW02] Roger S. Barga, David B. Lomet, and Gerhard Weikum. Recovery guarantees for general multi-tier applications. In *ICDE*, pages 543–554, March 2002.
- [BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Distributed systems. In S. Mullender, editor, *Distributed Systems*, ACM Press, chapter 8 – The primary-backup approach, pages 199–216. Addison-Wesley, second edition, 1993.
- [BMT02] R. Baldoni, C. Marchetti, and A. Termini. Active software replication through a three-tier approach. In *Proceedings of*

- IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002.
- [BR93] Kenneth P. Birman and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. IEEE Computer Society Press, 1993.
- [BvST98] Arno Bakker, Maarten van Steen, and Andrew S. Tanenbaum. Replicated invocations in wide-area systems. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 130–137. ACM Press, 1998.
- [CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems (3rd ed.): concepts and design*. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [Coo85] Eric C. Cooper. Replicated distributed programs. In *Proceedings of the tenth ACM symposium on Operating systems principles*, pages 63–78. ACM Press, 1985.
- [Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
- [DG01] Eliezer Dekel and Gera Gofit. Itra: Inter-tier relationship architecture for end-to-end QoS, 2001.
- [DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
- [DS02] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication and lazy consensus. Technical Report IC/2002/007, École Polytechnique Fédérale de Lausanne, Switzerland, February 2002.
- [DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 43. IEEE Computer Society, 1998.
- [DVDR01] Kaushik Dutta, Debra VanderMeer, Anindya Datta, and Krithi Ramamritham. User action recovery in internet sagas (iSAGAs). *TES*, 2193:132–146, 2001.

- [FCK87] Johann Christoph Freytag, Flaviu Cristian, and Bo Kähler. Masking system crashes in database application programs. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 407–416. Morgan Kaufmann, 1987.
- [FG99] Svend Frølund and Rachid Guerraoui. Transactional exactly-once, 1999.
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [FN02] Pascal Felber and Priya Narasimhan. Reconciling replication and transactions for the end-to-end reliability of corba applications. In *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002*, pages 737–754. Springer-Verlag, 2002.
- [GGM94] Benoît Garbinato, Rachid Guerraoui, and Karim Mazouni. Distributed programming in garf. In *Proceedings of the Workshop on Object-Based Distributed Programming*, pages 225–239. Springer-Verlag, 1994.
- [GMUW02] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The complete book*. Prentice-Hall, Inc., 2002.
- [GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [HBH96] Abdelsalam A. Helal, Bharat K. Bhargava, and Abdelsalam A. Heddaya. *Replication Techniques in Distributed Systems*. Kluwer Academic Publishers, 1996.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [LS98] M. Little and S. Shrivastava. Integrating the object transaction service with the web. In *Proceedings of the Second International Workshop on Enterprise Distributed Object Computing (EDOC), IEEE*, 1998.

- [LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [LY01] Mon-Yen Luo and Chu-Sing Yang. Constructing zero-loss web services. *INFOCOM*, pages 1781–1790, 2001.
- [MGG95a] K. R. Mazouni, B. Garbinato, and R. Guerraoui. Building reliable client-server software using actively replicated objects. In I. Graham, B. Magnusson, B. Meyer, and J.-M. Nerson, editors, *Proceedings of the TOOLS EUROPE'95 Conference*, pages 37–51, Versailles, France, 1995. Prentice-Hall.
- [MGG95b] Karim R. Mazouni, Benoît Garbinato, and Rachid Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proceedings of the 4th International Workshop on Object-Oriented Programming in Operating Systems*, page 118. IEEE Computer Society, 1995.
- [MMSA<sup>+</sup>96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
- [OMG03] Object Management Group. *Transaction Service Specification*, September 2003. OMG Technical Committee Document formal/03-09-02.
- [OMG04] Object Management Group. *Fault Tolerant CORBA*, March 2004. OMG Technical Committee Document formal/04-03-21.
- [PKS03a] S. Pleisch, A. Kupšys, and A. Schiper. Preventing orphan requests in the context of replicated invocation. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 119 – 128, Florence, Italy, October 2003. IEEE.
- [PKS03b] S. Pleisch, A. Kupšys, and A. Schiper. Replicated invocations. Technical report, Swiss Federal Institute of Technology (EPFL), September 2003.
- [Pol93] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. Research Report 6/1993, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.
- [Pow91] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Number 818/2252 in Research Reports,

- ESPRIT 818/2252. Springer-Verlag, 1991. ISBN 3-540-54985-4.
- [Sch93] Fred B. Schneider. *Replication management using the state-machine approach*, pages 169–197. ACM Press/Addison-Wesley Publishing Co., 1993.
- [TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.
- [VB98] Alexey Vaysburd and Ken Birman. The maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theor. Pract. Object Syst.*, 4(2):71–80, 1998.
- [VBB<sup>+</sup>91] P Veríssimo, P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta-4 - A Generic Architecture for Dependable Distributed Computing*, ESPRIT Research Reports, chapter 9, pages 211–266. Springer Verlag, nov 1991.
- [XA92] X/OPEN Company Ltd. *Distributed Transaction Processing: The XA Specification*, February 1992.
- [ZMMS02] W. Zhao, L. E. Moser, and P.M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 290–297, 2002.