NOTE TO USERS

This reproduction is the best copy available.

UMI®

ADAPTABLE STATEFUL APPLICATION SERVER REPLICATION

Huaigu Wu

DOCTOR OF PHILOSOPHY

the School of Computer Science

MCGILL UNIVERSITY

MONTREAL, QUEBEC

OCTOBER 2008

A THESIS SUBMITTED TO MCGILL UNIVERSITY IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF THE DEGREE OF DOCTOR OF PHILOSOPHY

COPYRIGHT BY HUAIGU WU 2009

©ALL RIGHTS RESERVED

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-66694-4
Our file Notre référence
ISBN: 978-0-494-66694-4

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my supervisor, Dr. Bettina Kemme, for her continued guidance, financial supports, encouragement and patience through all the phases of this research.

Extended thanks to Dr. A. Bartoli, Dr. V. Maverick, and all other research colleagues in the project "ADAPT" , for their lively and fruitful collaborations and discussions regarding this research.

My sincere appreciation to all my friends and colleagues whom I met in Montreal, Chengdu and Chongqing, for their joys and laughs.

My deepest appreciation to my family, especially to my wife, my parents, and my parents-in-law, who have been so supportive and loving. Without them, this report would not be the same.

Finally, I dedicate my thesis to my unborn baby with my love.

ABSTRACT

In recent years, multi-tier architectures have become the standard computing environment for web- and enterprise applications. The application server tier is often the heart of the system embedding the business logic. Adaptability, in particular the capability to adjust to the load submitted to the system and to handle the failure of individual components, are of outmost importance in order to provide 7/24 access and high performance. Replication is a common means to achieve these reliability and scalability requirements. With replication, the application server tier consists of several server replicas. Thus, if one replica fails, others can take over. Furthermore, the load can be distributed across the available replicas. Although many replication solutions have been proposed so far, most of them have been either developed for fault-tolerance or for scalability. Furthermore, only few have considered that the application server tier is only one tier in a multi-tier architecture, that this tier maintains state, and that execution in this environment can follow complex patterns. Thus, existing solutions often do not provide correctness beyond some basic application scenarios.

In this thesis we tackle the issue of replication of the application server tier from ground off and develop a unified solution that provides both fault-tolerance and scalability. We first describe a set of execution patterns that describe how requests are typically executed in multi-tier architectures. They consider the flow of execution across client tier, application server tier, and database tier. In particular, the execution patterns describe how requests are associated with transactions, the fundamental execution units at application server and database tiers. Having these execution patterns in mind, we provide a formal definition of what it means to provide a correct execution across all tiers, even in case failures occur and the application server tier is replicated. Informally, a

replicated system is correct if it behaves exactly as a non-replicated that never fails. From there, we propose a set of replication algorithms for fault-tolerance that provide correctness for the execution patterns that we have identified. The main principle is to let a primary AS replica to execute all client requests, and to propagate any state changes performed by a transaction to backup replicas at transaction commit time. The challenges occur as requests can be associated in different ways with transactions. Then, we extend our fault-tolerance solution and develop a unified solution that provides both fault-tolerance and load-balancing. In this extended solution, each application server replica is able to execute client requests as a primary and at the same time serves as backup for other replicas. The framework provides a transparent, truly distributed and lightweight load distribution mechanism which takes advantage of the fault-tolerance infrastructure. Our replication tool is implemented as a plug-in of JBoss application server and the performance is carefully evaluated, comparing with JBoss' own replication solutions. The evaluation shows that our protocols have very good performance and compare favorably with existing solutions.

ABRÉGÉ

Au cours des dernières années, l'architecture multi-tiers est devenue la norme pour le développement d'applications Web et d'entreprise. Dans cette architecture, le serveur d'applications représente souvent le cœur du système encapsulant la logique de traîtement. La capacité d'un tel système à s'adapter à la charge soumise et à gérer les défaillances des composantes individuelles sont d'une importance capitale afin de fournir un accès permanent et performant à l'application.

La réplication est un moyen très utilisé pour atteindre la fiabilité et l'extensibilité requises. Avec la réplication, le serveur d'applications dispose de plusieurs copies. Ainsi, si une copie ne parvient pas à répondre à une requête, les autres peuvent prendre la relève. En outre, la charge peut être répartie entre les copies disponibles. Bien que de nombreuses solutions de réplication aient été proposées, la plupart d'entre elles ont été conçues soit pour résoudre le problème de tolérance aux fautes, soit pour résoudre le problème d'extensibilité. En plus, seules quelques unes ont considéré le fait que le serveur ne représente qu'un seul niveau dans l'architecture multi-tiers et que l'exécution dans cet environnement peut suivre des patrons complexes. Ainsi, souvent les solutions existantes ne prévoient pas l'exactitude au-delà de quelques scénarios de base.

Dans cette thèse, nous abordons la question de la réplication du serveur d'applications ainsi que le développement d'une approche de réplication qui unifie la tolérance aux fautes et l'extensibilité du système. Pour adresser ce point, nous avons d'abord identifié un ensemble de patrons d'exécution qui décrivent comment les requêtes sont généralement exécutées dans les différents niveaux de l'architecture. Ces patrons considèrent le flux d'exécution à travers le client, le serveur, et la base de données. En particulier, ils décrivent comment les requêtes sont liées à des transactions et des unités d'exécution fondamentales au serveur d'applications et aux bases de données.

Ayant ces patrons, nous fournissons une définition formelle de l'exécution correcte dans tous les niveaux de l'architecture, même dans le cas où des défaillances se produisent et le serveur d'applications est répliqué. Officieusement, une réplication d'un serveur est correcte si elle se comporte exactement comme si le serveur n'a jamais fait face à des défaillances. De là, nous proposons une série d'algorithmes de réplication pour la tolérance aux fautes qui assurent l'exactitude des patrons d'exécution que nous avons identifiés. L'idée principale est de laisser une copie primaire exécuter toutes les requêtes des clients, et de propager tout changement d'état aux copies de sauvegarde quand la transaction est validée. Les défis résident dans le fait que les requêtes peuvent être associées aux transactions de différentes manières. Ensuite, nous étendons notre solution pour prendre en considération la tolérance aux fautes et l'équilibrage de charge entre les serveurs. Dans cette solution, chaque copie du serveur d'applications est en mesure d'exécuter les requêtes des clients comme une copie primaire et en même temps elle sert comme sauvegarde pour les autres copies. Cette plate-forme offre un mécanisme de répartition de la charge qui est transparent, distribué, léger et qui profite de l'infrastructure de la tolérance aux fautes. Notre outil de réplication est implémenté comme un plug-in du serveur d'applications JBoss. Les performances ont été évaluées avec soin, en les comparant avec les solutions de réplication JBoss. L'évaluation montre que nos protocoles ont de très bonnes performances et qu'elles dépassent celles des solutions existantes.

Contents

ACKNOWLEDGEMENTS ii

ABSTRACT iii

ABRÉGÉ v

List of Tables xii

List of Figures xiv

1 Introduction 1

1.1 Motivation 1

1.1.1 Replication for Fault Tolerance 2

1.1.2 Replication for Load Balancing 5

1.2 About This Work 6

1.2.1 Modeling Execution Patterns in a Multi-Tier Architecture 7

1.2.2 Development of a Replication Tool for Fault Tolerance 7

1.2.3 Extension of the Replication Tool to Support Load Balancing 8

1.2.4 Implementation of a Replicated Application Server 9

1.2.5 Contribution of this Thesis 10

1.3 Structure of the Thesis 11

2 Background 12

2.1 Overview of Multi-tier Architecture 12
2.2 Overview of Application Server Tier 13
2.2.1 Execution Flow 13
2.2.2 Calling Schemes 14
2.2.3 Server State 15
2.2.4 Determinism 15
2.3 J2EE Application Server 16
2.3.1 Enterprise JavaBean 16
2.3.2 Execution Flow 17
2.4 Transaction Management 18
2.4.1 Lifespan of a Transaction 18
2.4.2 State Consistency 20
2.4.3 Concurrent Transactions 21
2.4.4 Relation between Requests and Transactions 22
2.5 Overview of Failures 26
2.5.1 Failure Types 26
2.5.2 Crash of an Application Server 27
2.6 Tolerate Failures through Replication 28
2.6.1 Passive Replication Category 29
2.6.2 Correctness of Replication 31
2.7 Communication Mechanism for Replication 31
2.7.1 Group Membership Maintenance 32
2.7.2 Multicast 33

3 Traditional Application Server Replication Solutions and Correctness Criteria 35

3.1 Overview of Existing Replication Solutions for Fault Tolerance 35
3.1.1 Replication for J2EE Architecture 36
3.1.2 Replication for CORBA Architecture 37
3.1.3 Replication for .NET Architecture 39

3.2 Traditional Correctness Criteria for Replication 40
3.2.1 One Copy Serializability 40
3.2.2 State Machine Replication 41
3.2.3 X-ability 42
3.3 Load Balancing and Combined Approaches 44

4 Execution Patterns for Application Server **46**
4.1 Request Execution 47
4.1.1 Histories of Request Execution 48
4.2 Transactions and Execution Patterns 49
4.2.1 1-1 Pattern 49
4.2.2 N-1 Pattern 51
4.2.3 1-N Pattern 52
4.2.4 N-N Pattern 55
4.3 State Changes 56
4.3.1 Transaction Histories to Reflect the Order of State Changes 57
4.3.2 Matching State Changes at Application Server and Database 58
4.3.3 Matching State Changes at Application Server and Client Request Execution 59
4.4 Correct Request Execution 63
4.5 How a Crash Affects Correctness 64
4.6 Correctness of Passive Replication 65

5 ADAPT-SIB Replication Algorithm for 1-1 Pattern **66**
5.1 Structure of ADAPT-SIB 67
5.2 1-1 Replication Algorithm for Full State Consistency 69
5.3 Correctness 72
5.3.1 Successfully Completed Requests 73
5.3.2 Crash during Request Execution 74
5.4 1-1 Replication Algorithm for Relaxed State Consistency 79
5.5 Correctness of Relaxed State Consistency Algorithm 81

6 Advanced Algorithms for Advanced Execution Patterns **83**

6.1 N-1 Pattern 83

6.1.1 N-1-best-effort 84

6.1.2 Correctness of N-1-best-effort 88

6.1.3 Relaxed State Consistency 95

6.1.4 Increasing the Chances for Exactly-Once 96

6.2 1-N Pattern 97

6.2.1 Sub-requests and Nested Transactions 97

6.2.2 1-N Algorithm Overview 100

6.2.3 1-N Algorithm Details 102

6.2.4 Correctness Discussion 105

6.2.5 Undo Ghost Transactions 113

7 Miscellaneous Extensions of ADAPT-SIB **115**

7.1 Different Failover Strategies 115

7.2 Recovery 117

7.3 Non-transactional Client Requests 118

7.4 Accessing more than one Database 119

7.5 Client and Database Crashes 120

7.5.1 Database Crash 120

7.5.2 Client Crash 121

7.5.3 Replicated Database and Replicated Clients 121

8 ADAPT-LB: Load Balancing Architecture based on ADAPT-SIB **123**

8.1 Algorithm Overview 124

8.2 Cluster Initialization 126

8.3 Load Balancing Algorithm 127

8.3.1 Simple Load Distribution 127

8.3.2 Load Forwarding 128

8.3.3 Discussion 130

8.4 Reconfiguration 131
8.4.1 Server Crash 131
8.4.2 Server Recovery 132
8.4.3 Reconfiguration Effects on Client 133

9 Implementation **134**
9.1 J2EE Architecture 134
9.1.1 EJB Lookup 135
9.1.2 Interceptor Chain 136
9.1.3 Associating Transactions with Requests 137
9.2 Implementation based on the Adapt Framework 138
9.2.1 Components and States 139
9.2.2 Invocation Interception 140
9.2.3 Requests and Responses 141
9.2.4 Transaction Interception 142
9.2.5 JDBC Interception 144
9.2.6 Overall Architecture 144
9.3 Implementation Issues 145
9.3.1 Extended Naming Service for the Replicated Application Server 146
9.3.2 Deciding on the Primary 147
9.3.3 Processing Requests and Transactions 147
9.4 Summary 148

10 Experiments and Evaluation **149**
10.1 Evaluation of ADAPT-SIB 149
10.1.1 Performance Comparison between warm and cold Replication 150
10.1.2 Component Analysis 152
10.1.3 Evaluation of Different Execution Patterns 155
10.1.4 Evaluation of Failover 159
10.2 Evaluation of ADAPT-LB 162

10.2.1 Experiment 1: Basic Performance 163
10.2.2 Experiment 2: Scalability 165
10.2.3 Experiment 3: Heterogeneity 165
10.2.4 Experiment 4: ECperf Benchmark 168
10.2.5 Experiment 5: Reconfiguration 169

11 Conclusions and Future Work **173**

11.1 Summary 173
11.1.1 Correct Replication for Different Execution Patterns 173
11.1.2 Performance 174
11.1.3 Practicability 174
11.2 Future Work 175
11.2.1 Enhancement to Handle Shared Data 176
11.2.2 Replication across a WAN 176
11.2.3 Extension of ADAPT-LB 177

Bibliography **178**

List of Tables

2.1 Classification of passive replication of commercial products 30

10.1 1-1 execution accessing one or more than one database 159

List of Figures

1.1 Online shopping: an example of a web based application 2

2.1 Application server architecture 14
2.2 Execution flow in J2EE architecture 17
2.3 Code snippet of CMT 23
2.4 Code snippet of BMT 23
2.5 Code snippet of user transaction 24
2.6 Group communication system 32

4.1 Sample scenario of request execution 48
4.2 1-1 pattern 50
4.3 N-1 pattern 52
4.4 1-N pattern 53
4.5 N-N pattern 55

5.1 Architecture of ADAPT-SIB 67
5.2 1-1 Algorithm at the client and primary 69
5.3 1-1 failover 72
5.4 Possible crash intervals of the 1-1 algorithm in case of a commit 74
5.5 Possible crash intervals of the 1-1 algorithm in case of an abort during execution . . 77
5.6 Possible crash intervals of the 1-1 algorithm in case of an abort at commit 79
5.7 "1-1-relaxed" algorithm to support relaxed state consistency 80

5.8 Possible crash intervals of the relaxed state consistency algorithm in case of an abort during execution 81
5.9 Possible crash intervals of the relaxed state consistency algorithm in case of an abort at commit 82

6.1 N-1-best-effort at the client side 86
6.2 N-1-best-effort at primary 87
6.3 Possible crash intervals of the N-1 algorithm in case of a commit 89
6.4 Possible crash intervals of the N-1 algorithm in case of a commit 92
6.5 An example execution of the 1-N pattern 98
6.6 Ttree and Rtree 98
6.7 "1-N" algorithm 103
6.8 Possible crash intervals of the 1-N algorithm for sibling inner transactions in case of a commit 106
6.9 Possible crash intervals in case of an abort of an outer transaction 109
6.10 Possible crash intervals in case of an abort of an inner transaction 110
6.11 Possible crash intervals of the 1-N algorithm for nested inner transactions in case of a commit 111

8.1 Unified architecture of ADAPT-LB 125
8.2 Initial setting with $m = 2$ 127
8.3 Forwarding a request 129
8.4 Crash scenario 132
8.5 Recovery scenario 132

9.1 Lookup EJB from the client side 135
9.2 Interceptor chain 136
9.3 ADAPT framework separates replication algorithm from J2EE server 139
9.4 ADAPT framework intercepts the execution flow at three points 140
9.5 ADAPT framework wraps transaction manager and client-side user transaction . . 143

9.6 The implementation architecture of the ADAPT framework in JBoss 145

10.1 Performance comparison between warm and cold replication 150
10.2 No database access 153
10.3 Conflict-free database access 153
10.4 Conflicting database access 153
10.5 ECperf comparison for the "1-1" Pattern 157
10.6 ECperf comparison for the "N-1" Pattern 157
10.7 ECperf comparison for the "1-N" Pattern 157
10.8 Restore strategies comparison 160
10.9 Failover time for different running time of ECPerf at 5 IR 162
10.10 Performance improvement 163
10.11 Scale-up homogenous setup 164
10.12 Scale-up heterogeneous hardware 164
10.13 Throughput distribution 166
10.14 Response time distribution 166
10.15 Heterogeneous workloads 167
10.16 Scalability for ECperf benchmark 169
10.17 Reconfiguration: Failover 169
10.18 Reconfiguration: Recovery 169
10.19 Comparison of failover operations 171

Chapter 1

Introduction

1.1 Motivation

In recent years, with the rapid growth of the Internet, more and more enterprise applications are established using web technology. Typical web-based applications are using a 4-tier architecture, which consists of client tier, web server tier (WS), application server tier (AS), and backend database tier. In such an architecture, clients first send requests to a WS, which processes presentation logic, such as generating web pages. Then, the request is passed to an AS, which processes business logic (e.g., maintaining a shopping cart, executing a purchase operation, etc.) and accesses database systems to manage persistent data. WS and AS together are also called the middle tier.

Clients of the WS are usually the real clients of the application and are connected via the Internet. The client of the AS is the WS, and the AS is the client of the backend database. Both WS and AS can contain volatile state that can exist beyond the execution of individual requests. We say such systems are *stateful*. Figure 1.1 shows a typical example of a web based application, where a client buys a book online. The client submits the purchasing request through the web page of an online book store like Amazon. The request is first parsed on the WS and then passed to the AS. The AS processes the purchase that includes adding the book to the client's shopping cart and executing the payment. The records about the purchase and the payment are stored at the backend database. The request execution on the AS including the accesses to the database are typically transactional

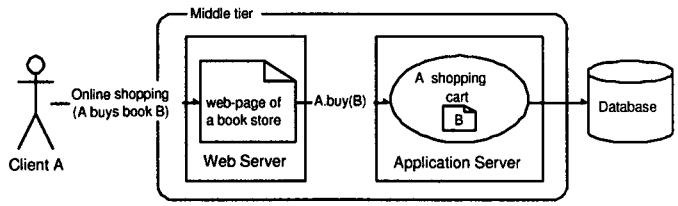


Figure 1.1: Online shopping: an example of a web based application

in order to provide durability for the persistent data, isolation from concurrent transactions, and atomicity. Nowadays, web-based applications, such as online stores, online banking, online games, and online communities, are growing very fast, involving people all over the world, and influencing almost all areas of our life.

The AS is the heart of the typical 4-tier architecture executing the kernel logic. The use of AS technology is growing very fast along with the increasing market of web-based applications. With 400 million web sites already in existence and growing, the need for AS is growing. As reported in [105], the AS market at $1.5 billion in 2003, is expected to reach $5.2 billion by 2009. In many web-based applications, the AS executes crucial and heavy loaded tasks, demanding to be accessible on a 7/24 basis and to provide short response time to users. Both AS's fast growing market and AS's vital position in web-based applications strongly require the AS tiers to be highly adaptable, in particular, to provide high reliability, availability, and scalability. Special challenges exist for the adaptability of stateful AS due to the difficulties of managing the volatile state at the AS.

Motivated by these requirements, this dissertation focuses on using replication to implement adaptable stateful AS. This dissertation studies fault tolerance to address reliability and availability, and studies load balancing to address scalability. Replication is used to provide extra resources to both tolerate failures and to distribute load.

1.1.1 Replication for Fault Tolerance

Replication is an essential mechanism to tolerate failures by allowing a system to have more replicas to backup data or actions. It can be used in every tier of a multi-tier architecture. The basic idea is

to let a tier have several replicas. When one replica of the tier fails, other replicas can take over and continue the work assigned to the failed replica. Replication can be either *active* or *passive*. In an active scheme, a request is sent to and executed at all replicas. When a replica fails, other replicas continue execution. In a passive scheme, only the primary replica executes the request, and other replicas backup the data changes executed at the primary replica. If the primary fails, one of the backups becomes the new primary to take over request execution.

Replication of the database tier has been well studied, e.g., [17, 104, 2, 24, 64]. Most DBMSs (database management system) already use replication to tolerate failures, such as Oracle Real Application Clusters (RAC), Microsoft SQL Server 2000 Failover Clustering, and IBM's DB2 replication solution. Replication is also widely used for middle tier systems, as proposed in, e.g., [29, 72, 73, 74, 46, 41, 114, 113, 11, 10, 62]. Both the prevailing WS products (e.g., Apache, Tomcat, and Microsoft IIS) and AS products (e.g., BEA WebLogic, IBM WebSphere, JBoss and Microsoft .NET) have their own replication solutions. Most middle tier systems use passive replication since it requires less resources and less management overhead, and allows for non-deterministic execution.

Nevertheless, replication of the middle tier still has many open questions. A crucial problem is how to guarantee correctness in a replicated middle tier system. To solve this problem, we have to clarify what correctness means for such a system. Informally speaking, correctness of a replication algorithm requires the replicated system to act in the same way as a non-faulty non-replicated system. However, the standard behavior of such a middle tier system is not clearly defined yet. Most of the existing solutions only assume quite simple semantics for request execution across the tiers. When an application does not follow the basic execution model, corresponding replicated systems might expose incorrect behavior. So far, however, only few approaches (e.g., [11, 114]) consider the more complex execution models that often occur in real systems.

This problem is very severe for the AS tier. The business logic processed at the AS might be very complex. A particular challenge is that the execution of client requests at AS and database can be associated with transactions in different ways. The simplest association is that each client request executes within an individual transaction. That is, all read and write operations on data residing on the AS and/or the database take place in a single transaction. For example, as shown in figure 1.1,

when a WS receives a request to buy a book from a web client, it submits only one request to the AS to be executed within a single transaction. Most of the existing solutions assume this simple association. In practice, however, execution can be more complex. At the one extreme, the AS's client, namely the WS, can start a transaction, and then submit several requests in the context of this transaction before committing it. For instance, within a purchase transaction, the WS might submit several requests to the AS to retrieve the book and make the payment. At the other extreme, a client request might create several independent transactions. For example, application programmers often chop the execution of a request into a set of small transactions to avoid lock contention at the database. Existing solutions will simply not work correctly if an application follows such advanced execution models. As a result, there is a large gap: the AS required to be replicated is usually used in critical environments with heavy load and high possibility of failures, but existing solutions cannot guarantee correct fault tolerance for the AS under such circumstances. In order to bridge this gap, this dissertation analyzes execution patterns that are used in practice to associate requests and transactions and proposes a set of replication algorithms, each of which provides the correct replication semantics for a different pattern. Our algorithms follow the passive scheme, i.e., a primary executes requests, backups can take over in the failure case.

Another major challenge is how to prove the correctness of these algorithms. So far, much of the research on correctness of replication of middle tier systems looks at specific aspects by considering specific replication abstractions (e.g., relation between replication and failure types [22], total order broadcast of requests [88], or consensus [25, 31]), but falls short of considering the global picture (different replicated systems, different application semantics, interaction with other tiers, etc.). Only a few approaches (e.g. [44] and [33]) present a formal correctness criteria for replication of middle tier systems. However, these criteria are usually built on some specific assumptions or for some specific environments. Hence, another effort of this dissertation is to define a suite of correctness criteria for AS replication, and use it to prove the correctness of our replication algorithms.

Except for correctness, our replication algorithms also address two usability requirements. First, the replication of the AS tier should not require any special support from other tiers. Ideally, the other tiers are not even aware of the fact that the AS tier is replicated. In this way, the replicated AS will be general enough to connect with different kinds of WS and backend DBMSs without

changing the standard interaction models defined in the corresponding specifications. Thus, no changes to the other tiers are needed. Secondly, since different execution patterns might be mixed in an application, different replication algorithms need to be supported concurrently to automatically adapt to the different execution patterns at runtime.

1.1.2 Replication for Load Balancing

Replication is also the major approach to implement load balancing. It increases the scalability of a system by distributing the load across the replicas of the system. Typical load balancing solutions for AS (or WS) use a centralized load balancer (also called scheduler) to manage the load dispatch. A task is first sent to the load balancer. Then, the load balancer uses a *content-blind policy* or a *content-aware policy* [4] to choose replicas to dispatch loads. In content-blind policies, such as *Random* or *Round Robin*, the load balancer does not know the load on each site. These policies can be easily implemented and hence, are widely used in practice, especially in most AS products, such as BEA WebLogic, IBM WebSphere, and JBoss. However, they cannot work well when the workload is diverse, or the system is heterogenous. A content-aware policy means the load balancer has knowledge about the load on each replica, e.g., the CPU and memory resource utilization or the response times. The load balancer can use the knowledge to optimize the load distribution and orchestrate resources of all replicas.

Load balancing strategies are pervasively used in many enterprise applications, especially in critical environments, where heavy load has to be processed. However, a crucial problem is that a replication architecture should not only support load balancing, but also fault tolerance, since such a critical environment is usually failure prone as well. In particular, this is a vital requirement for the AS tier, since the core business logic is processed at the AS. In order to fulfill this requirement, an intuitive way is to build a combined replication architecture that supports both load balancing and fault tolerance. However, while replication has been separately studied and applied widely for both issues for a long time, only little research has been performed on providing a combined replication solution to handle both in a single architecture.

This dissertation proposes a unified replication architecture, where a cluster of AS replicas is used to balance load and tolerate failures. The main challenge to implement such a combined

replication architecture is that the mechanisms to use replication respectively for scalability and fault-tolerance are different and even conflicting, although both have a cluster of AS replicas. In order to achieve scalability, load balancing algorithms use AS replicas as resources to execute client requests. Ideally, the more replicas the cluster has, the higher the maximum throughput it can achieve. In contrast, fault-tolerance algorithms use AS replicas as redundant resources that can mask the failures of individual replicas. In both the passive and the active scheme, different replicas have the same data or execute the same tasks. Apparently, the redundancy decreases the scalability. Fortunately, the failure probability of an individual replica is low, and hence having two or three running replicas are enough for most applications. In a passive schema, since tasks performed by a backup typically require much less resources than executing requests itself, resources at backups might often be wasted. Our approach does not waste resources as it lets each replica be a primary for some clients executing requests and be a backup for some other servers at the same time.

A further challenge of load balancing strategies is the tradeoff between a precise load distribution and the overhead of implementation, maintenance and management of the load balancing strategy. As mentioned before, a content-aware policy can provide precise load distribution using the knowledge of the load at different replicas. However, the exchange and the maintenance of load knowledge among different replicas normally requires a significant overhead. The more precise the load distribution is, the higher overhead the system needs to pay. Moreover, a centralized load balancer is a stateful single point of failure and requires extra replication overhead for fault tolerance. A distributed load balancer is rarely used for AS because it is too complex to be implemented. This dissertation addresses this issue by building an effective yet simple distributed load balancing algorithm.

1.2 About This Work

The work of this thesis is part of the "ADAPT" (Middleware Technologies for Adaptive and Composable Distributed Components) project, which is interested in developing support for the creation of adaptable web services. Partners involved in this project are Universitá di Bologna, Universitá di Trieste, Universidad Politécnica de Madrid, ETH Zurich, Universitá di Trieste, University of

Newcastle, HP Arjuna Labs, and McGill University.

In this context, my thesis focuses on replication solutions for AS, addressing fault tolerance and load balancing. The following sections provide a detailed overview of the contributions.

1.2.1 Modeling Execution Patterns in a Multi-Tier Architecture

Informally speaking, a correct replication algorithm should guarantee that the replicated system, despite the possibility of failures, works in the same way as a non-faulty non-replicated system. Hence, to define the correctness criteria to be supported by the AS replication algorithm, we need to model the behavior of a non-faulty non-replicated AS in a multi-tier architecture as the standard for correctness. We use *execution patterns* to model the relationship between requests and transactions. Due to the multi-tier architecture, the failure and the replication of the AS might not only affect the AS itself, but also affect the client tier of the AS and the database tier linked to the AS. Hence, the analysis has to take the client tier and the database tier into account.

The simplest execution pattern, referred as "1-1" pattern, indicates that each client request executes within its own individual transaction. The N-1 pattern associates a transaction with more than one client request, and the 1-N pattern associates a client request with more than one transaction. For each execution pattern, we analyze different transaction termination behaviors. Then, we model a failure of an AS by analyzing the side effect of the failure. To simplify the problem, the thesis currently only focuses on crash failures.

Finally, we model the replicated AS, and formally define correctness criteria for AS replication. Considering most practical AS products use passive replication schemes, our correctness criteria currently only focuses on passive replication schemes for simplification. However, we believe that similar mechanisms can be used to describe other schemes, such as active replication.

1.2.2 Development of a Replication Tool for Fault Tolerance

Although there exist some research using active replication (e.g., [69, 7, 35, 72]), and some considering a combination of active and passive replication (e.g., [31, 32]), most practical solutions for middle tier replication (e.g. [51, 45, 43, 41, 73, 11, 62]), especially those of commercial systems,

use passive replication. Our replication tool also uses the passive approach.

In our solution, called *ADAPT-SIB*, a server replica is the primary executing client requests, and other replicas are backups. The primary propagates state changes to the backups whenever a transaction commits. If the primary fails, a backup replica fails over, reconstructs the state of the old primary, and continues the client connections. Requests that were active at the time the primary crashed are automatically restarted at the new primary. The resubmission of requests is automatically done by a special stub at the client side that is implicitly downloaded from the server side without affecting the original client program. When requests are reexecuted after a crash, their sub-requests to the database are required to be coordinated with original transactions before the crash. To do so, we use a special agreement protocol using a marker mechanism similar to [43]. Unlike other coordination mechanisms, which either require additional support from the database like [11] or change the interface between AS and database like [114], our coordination mechanism does not require any additional support from the database, and uses the most common interface to access the database. It also differs from traditional agreement protocols like 2-phase-commit [65] since it has a highly reduced logging cost and does not require all participants to have executed the request before terminating. As a result, our replication solution guarantees independence to other tiers. We first design the replication algorithm for the simplest 1-1 pattern (published in [109]). Then, we extend the algorithm to support all execution patterns (published in [106]) and different transaction termination behaviors.

Besides above main issues, the work of the thesis also includes miscellaneous technologies associated with replication, such as designing and implementing a recovery strategy to allow failed nodes to recover and rejoin the system.

Moreover, when designing the algorithm, we always keep performance in mind. In particular, we address strategies to speedup failover. Our performance analysis shows that the approach compares favorably with other fault-tolerant solutions during normal processing, and has a fast failover.

1.2.3 Extension of the Replication Tool to Support Load Balancing

Based on the above ADAPT-SIB replication tool, we build an innovative AS replication solution to provide load balancing and fault tolerance in a unified architecture called *ADAPT-LB*. Unlike

ADAPT-SIB that just needs two or three replicas to support fault tolerance, ADAPT-LB can contain a large number of replicas. The entire cluster of replicas constructs a single *load distribution group* (LDG), where each member is a primary replica for some clients, executing the requests of this subset of clients. At the same time, each replica is backup for some other replicas. We refer to the group of one primary replica (executing requests) and the replicas that are backups of this primary as *fault tolerance group* (FTG). Thus, each replica is the primary of a small FTG and is backup in few other FTGs. As backup activity requires only few resources, the main capacity of each server is used for request execution.

The system uses a truly distributed, lightweight load-distribution algorithm that takes advantage of the existence of FTG groups. It does not require the maintenance of load information and keeps communication overhead for load-balancing purposes low. When a replica joins the system, it joins the LDG and creates a new FTG for which it is primary. When a replica fails or is removed from the system, a backup replica takes over its tasks. As part of any join or leave operation, the FTG configuration is adjusted to guarantee that all FTGs have a sufficient number of replicas and no replica is overburdened with backup tasks.

The load distribution algorithm combines the content-blind and content-aware policies. This way, the load balancing strategy can automatically adapt to the simple and low-overhead content-blind policy in a homogeneous environment, and switch to the content-aware policy to achieve precise load distribution in a heterogeneous environment. Furthermore, the load-balancing module will quickly remove any load imbalance that might occur during reconfiguration.

1.2.4 Implementation of a Replicated Application Server

To make our replication tool practical, we have implemented our replication tool within the context of a concrete AS architecture, namely J2EE [94] and integrated it into the open-source AS JBoss [49]. We choose the J2EE architecture because it is used very widely and it has many open-source products. We believe, however, that the principle ideas can be applied to other kinds of AS architectures (e.g., CORBA, .NET), and hence, we keep the algorithmic description as general as possible.

The implementation consists of several parts. The implementation of the suite of ADAPT-SIB replication algorithms is within a single replication package. The proper replication algorithm is

dynamically chosen at runtime according to the execution pattern used. The replication package is not directly linked to the JBoss environment. Instead, it is built on top of the ADAPT replication framework [6], whose implementation was a joint effort of our partners from Universitá di Bologna and Universitá di Trieste and us.

The ADAPT framework is an extension of a J2EE server, allowing replication algorithms to be plugged in. On the upper side, the framework defines a set of APIs for the replication algorithm to get state information and get control over requests and transactions. The replication algorithm can be implemented using these APIs without considering the architecture of a certain AS. Underneath the API, the framework has to implement the provided functionality. The implementation is highly compatible with underlying AS. In our implementation, the framework is based on the JBoss environment, the Java transaction APIs, and JDBC 2.0. Hence, our replication tool can be easily integrated into JBoss through the ADAPT J2EE replication framework.

To use ADAPT-SIB and ADAPT-LB for other J2EE products, we just need to change the ADAPT framework without any change to our replication algorithm itself.

1.2.5 Contribution of this Thesis

In summary, this thesis makes three main contributions.

- *Modeling*: The thesis defines a formal model to describe the behavior of a non-faulty non-replicated AS in a multi-tier architecture. The model helps to analyze the correctness of an AS replication algorithm, taking execution patterns into account.
- *Performance*: The thesis proposes replication protocols to support fault tolerance with good performance, i.e., about 15% extra overhead compared to a non-replicated AS. The load balancing strategy proposed in this thesis significantly increases the scalability of the replicated AS for both homogeneous and heterogeneous environments.
- *Practicability*: The thesis proposes a replication tool as a pluggable module that can be easily deployed and managed on different AS products without affecting clients (WS) and databases. The load balancing strategy uses a truly distributed, lightweight load-distribution algorithm that can be easily implemented and managed.

1.3 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 introduces some background in regard to AS replication, including the structure of application server, the execution patterns, failures, generic replication approaches, and the communication tool used for this thesis. Chapter 3 discusses related work. Chapter 4 describes the behavior of AS using execution patterns and defines the general correctness criteria for AS replication based on these patterns. Chapter 5, 6 and 7 present the replication tool for fault tolerance, called *ADAPT-SIB*. Chapter 5 presents the basic replication algorithm for the 1-1 pattern and discusses the correctness. Chapter 6 presents advanced replication algorithms for advanced execution patterns. Chapter 7 presents special features and extensions. Chapter 8 presents the extended unified replication framework called *ADAPT-LB* that supports both load balancing and fault tolerance. Chapter 9 presents the implementation of the replication tool for the JBoss Application server. Chapter 10 presents a thorough evaluation of the replication tool for both fault tolerance and load balancing. Chapter 11 concludes the thesis and discusses future work.

The replication algorithms proposed in Chapters 5, 6 and 7 have been previously published in [109, 106]. The correctness criteria of replication proposed in Chapter 4 was guided by our paper [107] that describes a formal model for reasoning about correctness of replication in 3-tier architectures. The ADAPT framework is described in [6]. A demo of the system was given in [108]. The unified replication framework proposed in Chapter 8 has been submitted for publication.

Chapter 2

Background

This chapter provides an overview of all basic concepts and terminology that will be used throughout the thesis. It introduces the concepts of multi-tier architecture, failure, replication, and a very useful communication mechanism to do replication, namely group communication.

2.1 Overview of Multi-tier Architecture

The multi-tier architecture is an extension of the traditional client-server computing model. The client resides outside the server, and only sees the service interface. It is the front tier, which usually directly interacts with end users and does not receive requests from other tiers. On the server side, the all-in-one server is split into multiple tiers. The backend tier is usually a database, which only receives requests from other tiers, but does not submit further requests to other tiers. Between these two tiers, there could be one or more middle tiers. When a middle tier receives a request from a preceding tier, the preceding tier is its client. Reversely, when the tier makes calls to another tier, it becomes the client of the called tier. A middle tier might be a client to more than one tier, and also be a server to more than one client. A multi-tier architecture separates an application into a set of building blocks. Each tier can be implemented as a self-contained component and deployed onto a separated machine.

An important benefit of a multi-tier architecture is that each tier can be designed and maintained separately without affecting the functionality of other tiers. It decreases the development and

maintenance cost of applications. Another advantage is that the overall overhead of an application is distributed across different tiers deployed on different machines. Although the corresponding communication overhead increases, the much heavier overhead to process application logic is distributed, and hence the overall performance can become better. Moreover, the performance at each tier can be fine-tuned separately, and hence better performance can be reached much easier.

2.2 Overview of Application Server Tier

Web based applications are typical use cases of the multi-tier architecture. The most prevalent middle tier systems used in these applications are WS and AS. In industry, there are three mainstream specifications to build the AS tier: J2EE [94], CORBA [77], and .NET [70]. Usually, in an AS, the business logic is modularized into different components that can call each other. Components can have state. All components of the AS are running within the same environment. Clients of the AS are usually the WS tier but also normal client programs. Backend servers of AS are usually databases. In some large enterprise applications, e.g. B2B applications, web services provided by different companies can be integrated into composite services according to some protocols such as UDDI, SOAP, and WSDL. In these applications, the AS also could call other WS or AS. But this is out of the scope of this thesis.

2.2.1 Execution Flow

Clients trigger business logic of the AS by submitting requests to the AS. When a client submits a request to the AS, the request typically calls a component method with input parameters. During the execution of the request, it may change the state of the component, and/or submit sub-requests to other components in the same AS or to the backend database. Each sub-request also may change the state of the component or the database, and/or submit further sub-requests. At the end of the execution of the request, the client will receive a response from the AS. The communication between clients and AS is usually based on existing protocols, like JAVA RMI, CORBA ORB, or Microsoft COM/DCOM. The communication between AS and database is usually based on database connection drivers provided by different DBMS products.

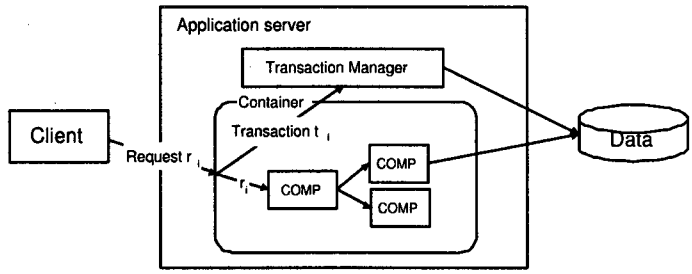


Figure 2.1: Application server architecture

Within the runtime environment, the AS also provides a set of services like transactions, persistence, security, messages, and database connection to support business logic. Most AS architectures provide two ways to access such services. Either components explicitly make service calls from programs or such service calls are made automatically by the runtime environment whenever a method of a component is activated. In the later case, the runtime environment can call the services before or after requests are executed. Figure 2.1 shows the execution flow of a client request within the AS with a transaction service being called.

2.2.2 Calling Schemes

When a client submits a request to an AS, the execution on the client is usually blocked until the response of the request is returned. This calling scheme is called *blocking scheme*. This is in contrast to the *non-blocking scheme*, which usually applies message queues. In here, after submitting a request, the client continues its execution without waiting for the response. In the AS, client requests are queued and processed in turn. The caller usually uses a listener to intercept the response, which triggers corresponding actions then. In this thesis we focus on the blocking calling scheme, because this is the typical model that are used by many communication protocols between clients and AS, e.g., JAVA RMI and CORBA ORB.

2.2.3 Server State

The state of an AS is the union of the states of all components of the AS. The state of an AS usually remains volatile in memory. Some AS products provide persistence mechanism for component state by temporally storing it on their local disk. This persistence is usually used to extend the main memory of the machine of an AS during runtime, but does not guarantee the durability in the same way a database would do. If the AS stops or crashes, the state stored on the local machine is discarded and no more available, even after the AS resumes work. Hence, in any case, we can think of the state of the AS as volatile.

In many cases, especially for the blocking scheme, when an AS receives requests from its clients, each client will build an individual session with the AS. In this case, much of the volatile state of an AS is *session-related.* Session-related state is only available for a certain client session, and is not shared by different sessions or different clients. Hence, no concurrency issues occur on session-oriented state, since concurrent requests of different clients access different state information.

However, there is state called *shared state* that can be concurrently accessed by different clients and different requests. The AS requires some concurrency control mechanism to reconcile potentially conflicting access to shared state. Different isolation levels are possible. Details are discussed in Section 2.4.3.

2.2.4 Determinism

Here, *determinism* means that if an AS runs two identical requests based on the same initial state, the responses and the state changes generated by the two requests are same. In particular, determinism is hard to achieve, even if we do not allow any non-deterministic programming model, e.g., multiple threads or time events are forbidden. This is because a very common cause for non-determinism are exceptions. An exception might be caused by many reasons, e.g., memory leak, system overload, program defect, or application semantics. If an exception in the backend database is not specially handled, it might be returned to the AS to cause non-determinism at the AS, and then be passed to the client. Accordingly, non-determinism can be passed from tier to tier. Hence, in this thesis, we

always keep non-determinism in mind.

2.3 J2EE Application Server

In recent years, J2EE has become the most popular specification to implement application servers. This section looks in detail how a J2EE based AS works. Taking a J2EE based AS as an example will help us to understand the practical issues relevant to AS replication.

2.3.1 Enterprise JavaBean

In a J2EE application server, components that implement business logics are special Java objects called *Enterprise JavaBeans (EJB)* [100]. We distinguish two categories: *session beans* (SB) and *entity beans* (EB)[1]. A session bean is a non-persistent object that represents the actions associated with a caller session. There are two subtypes. *Stateless session beans* (SLSB) do not maintain any internal state across method calls. *Stateful session beans* (SFSB) maintain internal state for the lifetime of a caller session. Obviously, SFSBs are components that have volatile state, and the state of a SFSB is the typical session-related state. The J2EE specification provides a passivation mechanism to transfer the in-memory volatile state of an SFSB instance to the storage managed by the AS, allowing the SFSB instance to be garbage-collected or reused. As we mentioned before, although this mechanism makes the state of an SFSB instance persistent, it is not designed for fault tolerance purposes, since when the AS crashes the persistent storage is not accessible either.

In contrast, an entity bean (EB) is an object that represents persistent data in persistent storage (mostly a database system). The state of an EB is the typical example of a shared state since different requests from different caller sessions can access the same EB. Also, the state of an EB represents cached data from the database. An EB synchronizes its state with the data in the database (i.e., read from or write to the database) within the boundaries of transactions. Thus, the concurrency control mechanism of the database can be used to manage the concurrent access of EBs in the cache. Further details in regard to the shared state follow in Section 2.4.3.

[1] Message beans are a third kind of EJBs. They are outside the scope of this dissertation.

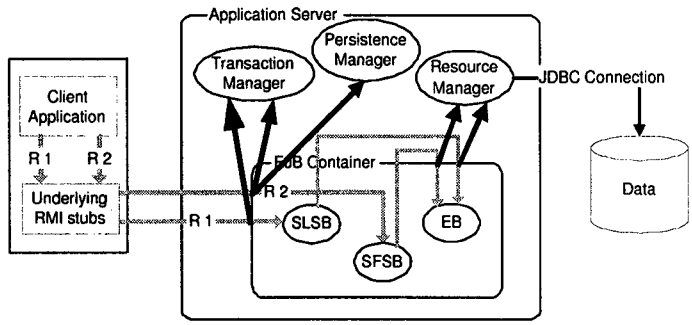


Figure 2.2: Execution flow in J2EE architecture

2.3.2 Execution Flow

At the client side, a client accesses EJB objects using the Java remote invocation (RMI) protocol. In order to access an EJB object, the client needs to first create a connection session to the AS. From this connection session, the client gets a remote reference (called *stub*) of an EJB object. Then, the client takes advantage of the stub to send requests to the server side EJB. The implementation detail of this procedure is described in Section 9.1.1.

At the server side, the runtime environment where EJBs are running in a J2EE AS is called *EJB container*. Whenever an outside client or an inside EJB makes a request to another EJB, the request will first be intercepted by the container, and then be dispatched to the destination EJB. The container is a wrapper of EJBs, which connects EJBs with services provided by the AS. It implicitly calls services according to the configuration, thus eliminating the need to code them within the application programs. To do so, the container intercepts requests to EJBs and calls certain services before requests are really executed, or after requests are executed but before the corresponding responses are returned to the caller. Figure 2.2 shows how an outside client submits two requests to an SFSB and an SLSB and how the container intercepts the two requests to call certain services.

Often, more than one service is required to be called in regard to a request. Hence, an EJB container usually consists of several interceptors, each of which is responsible for calling a certain service. All these interceptors form an interceptor chain (more on this in Section 9.1.2). Regarding the execution of a client request, the most relevant interceptor is the *transaction interceptor* that associates transactions with request executions.

2.4 Transaction Management

Transaction management is a key service for both AS and database. In an AS, requests are usually executed within the scope of a transaction. There also exists the possibility that a client request is executed without the context of a transaction in case there is no database access. However, for the sake of generality, we assume that all client requests have to be executed within the contexts of transactions. For a request that is not assigned to a transaction, we assume that a pseudo transaction covers its execution.

A transaction is a logical unit of read and write operations on component and database state. The well-known transactional properties are atomicity, consistency, isolation, and durability. In an AS, the transaction management is implemented by a transaction manager (TM). Figure 2.1 shows a typical way to call the transaction service. In the example, when the client request r_1 is passing the container, the container calls the TM to immediately trigger the start of a transaction t_1, then all execution related to r_1 happens within the boundaries of this transaction. That is, both the accesses to the three components in the AS and the access to the database triggered by r_1 are executed within the transaction t_1. Typically, each transaction is associated with a single thread in the AS and the execution of operations of the transaction is serial.

2.4.1 Lifespan of a Transaction

Typically, a transaction spans both execution on the AS and on the database. In practice, during runtime, such a *global transaction* consists of two parts: a transaction on the AS (called *AS transaction*) and a transaction on the database (called *DB transaction*), since the AS and the database are two different systems and have their own transaction management systems. Usually a global transaction identifier is used to identify that an AS transaction and a database transaction belong to the same global transaction.

Start of a global transaction

We can consider the lifespan of a global transaction as follows. Immediately after the TM executes a transaction begin request, a global transaction is started, and the corresponding AS transaction is

started at this time point. The corresponding DB transaction is only started when a sub-request is submitted to access the database during the AS transaction. If no sub-request accesses the database, the global transaction only contains the AS transaction but does not contain a DB transaction. After a global transaction is started, all state changes on AS and database triggered by the thread associated with the global transaction belong to this transaction (unless the transaction is suspended).

Termination of a global transaction

When a global transaction wants to commit, the TM executes a transaction commit request. During the execution, the TM submits a commit sub-request to the database. Only after the database successfully executes the commit request and returns the response to the AS, the AS can successfully finish the commit process. At this time point, the global transaction has committed on both the AS and the database, meaning both the corresponding AS transaction and the corresponding DB transaction have successfully committed.

A global transaction could be aborted. If an abort is triggered on the AS, the process is similar to the commit. That is, the TM executes the corresponding transaction abort request, sending an abort sub-request to the database. However, an abort can also be triggered by the database (e.g., deadlock, integration constraint violation). In this case, the database directly aborts the corresponding DB transaction, and then notifies the AS. After the AS receives the notification, the TM executes the abort request without sending an abort sub-request to the database.

Clearly, in all cases, a global transaction is first started on the AS as an AS transaction. Then, when persistent data is accessed, the corresponding DB transaction is started. No matter whether a global transaction eventually commits or aborts, its DB transaction always terminates before the termination of the corresponding AS transaction.

Accessing more than one database

In some applications, a transaction accesses more than one database. In this case, a 2-phase commit protocol (2PC) is necessary at commit time for atomicity [18]. The TM of the AS is the coordinator.

It first sends a *prepare* request to all participating databases which return either with a *prepared* message or their decision to abort. If all databases have successfully prepared, the TM sends a *commit* decision to all databases, otherwise (at least one aborted) an *abort* confirmation. The databases terminate the transaction accordingly.

2.4.2 State Consistency

The behavior of a DB transaction is clearly defined by the transaction properties. If a DB transaction commits, the state changes performed by the transaction on the database are made persistent. If the transaction aborts, any changes performed so far on the database are undone by the database system. In contrast, at the AS, the behavior depends often on whether it is shared or session state. Shared state is usually transactional by making it persistent. That is, at commit time the latest in-memory shared state is written to the database. In case of abort, changes are either undone or discarded and the state is reloaded from the database. In the following, we assume that shared state is always synchronized in this way with the database.

Most AS products do not provide durability for session-related state. Instead, changes on session-related state remain volatile. Moreover, some AS products do not provide atomicity for session-related state. That is, the abort of an AS transaction does not automatically trigger that changes on session-related state are undone. However, programmers can provide rollback methods to undo these state changes to guarantee atomicity. For example, in J2EE, programmers can define rollback methods for stateful EJBs. These methods are automatically called by the J2EE server in the abort case. We say the AS server provides *full state consistency* if mechanisms exist to abort changes on session-related state, otherwise it provides *relaxed state consistency*. When a global transaction aborts in case of relaxed state consistency, the AS might keep some state changes that have already been done so far or make some further changes before request execution finishes, while the database always aborts state changes of the corresponding DB transaction and is correctly rolled back in the abort case.

2.4.3 Concurrent Transactions

The AS might execute requests from different clients concurrently, leading to concurrent transactions, since requests from different clients have to run within different transactions. Accordingly, there can be concurrent AS and DB transactions.

On the database, a DBMS normally has well-established concurrency control mechanisms that can guarantee the isolation of concurrent transactions. In this dissertation, we assume that the DB provides serializability, which means that the execution of a set of transactions is equivalent to some serial execution of these transactions.

On the AS, as mentioned in Section 2.2.3, no concurrency control mechanism is required for session-related state since they are naturally isolated. However, a concurrency control mechanism has to be used to isolate accesses of concurrent transactions to shared state. There are two main alternatives. One option is that the AS has its own concurrency control mechanism; traditional approaches such as optimistic or pessimistic concurrency control could be used. Alternatively, the AS relies on the concurrency control mechanism of the database system. In our model, all shared state can be considered cached database state. Thus, the following approach can be taken. When an AS transaction wants to read a certain shared object for the first time it loads it from the database. Assuming locking, the corresponding DB transaction has a shared lock on this data item in the database. If now a concurrent transaction wants to write the data item, it also goes to the database and will be blocked in the database since the requested write lock conflicts with the existing read lock. The level of isolation is then the one provided by the DBMS. While having concurrency control at the AS level allows caching across transaction boundaries, relying on the DBMS for isolation only allows for intra-transaction caching. Whenever a transaction accesses a data item for the first time it has to go to the database. Clearly, the first approach is likely to be more efficient, and thus, more desirable as it allows for inter-transaction caching.

When we now consider the replicated case, in a purely fault-tolerant architecture where one primary AS replica executes all client requests, both concurrency control options are feasible, because then the concurrency control mechanism at the primary AS will serialize all transactions. However, in the case of load-balancing, several AS replicas execute client requests concurrently. In this

case the concurrency control modules of the different AS would need to coordinate. In contrast, if all the concurrency control is delegated to the DBMS it remains centralized at the DBMS and no extensions to the isolation infrastructure are needed.

As our solution aims at both fault-tolerance and load-balancing, we assume from now on that all shared state is synchronized with the database and concurrency control is only performed at the DBMS level. However, if our approach is used only for fault-tolerance, then the AS concurrency control method can be used without interfering with our replication mechanism.

In practice, most J2EE application servers offer the option to leave isolation of EBs to the DBMS. Each EB maps to a data record in the database. A transaction reads the up-to-date state of an EB from the database the first time it accesses the EB. Changes to the EB are written to the database and synchronized at commit time.

2.4.4 Relation between Requests and Transactions

In an AS, requests can be associated in different ways with transactions. *Execution patterns* are used to describe the association. We classify execution patterns by the number of client requests involved in a transaction and the number of transactions generated by a request. In the simplest execution pattern, a client request executes within its own individual transaction. All further sub-requests that are triggered by the client request to access other components in the AS and to access the database are also executed within the transaction. This basic execution pattern is called *"1-1" pattern* (1-request/1-transaction). But execution patterns could be more complex. In particular, J2EE allows a wide range of association of requests with transactions.

Transaction management at AS

In a J2EE AS, a transaction could be *Container-Managed* or *Bean-Managed.* In the container managed transaction (CMT) scheme, the EJB container has a transaction interceptor that intercepts each request and decides how to associate the execution with a transaction. Figure 2.3 shows a sample code snippet of the CMT scheme. In the CMT scheme, if a request is required to be executed within a new transaction, the transaction interceptor sends a transaction request to the TM to start a

```
TransactionInterceptor{
    ...
    Response invoke (Request req,
    Component comp){
        ...
        if (condition-1){
            TM.begin();
            comp.invoke(req);
            TM.commit();
        }
        ...
    }
}
```

Figure 2.3: Code snippet of CMT

```
Sample EJB object{
    ...
    void sampleMethod(){
        ...
        TM.begin();
        call an EJB method;
        update the database;
        if (condition-2)
            TM.commit();
         else {
            TM.abort();
            roll back changes;
         }
        ...
    }
}
```

Figure 2.4: Code snippet of BMT

transaction. If the execution successfully completes, the transaction interceptor sends a transaction commit request to the TM to commit the transaction, and only then returns the response. J2EE provides a set of options to decide where and when transactions are started. Each method of an EJB has a *transaction attribute* with the possible values: `Required`, `RequiresNew`, `Mandatory`, `Supports`, `NotSupported`, and `Never`. The most useful and popular attributes are `Required` and `RequiresNew`. The `Required` attribute means that an EJB method should be executed within a transaction. If a transaction is already associated with the current execution thread the method is executed within the existing transaction, otherwise a new transaction has to be started for the execution. The `RequiresNew` attribute means that an EJB method should be executed within a new transaction no matter whether a transaction already exists or not.

In the bean managed transaction (BMT) scheme, the code of a session bean[2] explicitly marks the boundaries of a transaction within an EJB method as shown in Figure 2.4. In this case, a transaction is only started during the execution of a request. Furthermore, during the execution, more than one transaction can be started one after another. According to the assumption that each request should be executed within the context of a transaction, we can assume a pseudo top-level transaction to cover the execution of each client request in the BMT scheme. Thus, each real transaction triggered within the method execution is kind of nested within the pseudo transaction.

[2]An Entity Bean cannot have Bean-Managed transaction according to the EJB specification [99]

```
...
UserTransaction.begin();
call EJB1.method1();
call EJB1.method2();
call EJB2.method1();
    ...
if (condition-3){
    UserTransaction.commit();
else
    UserTransaction.abort();
    ...
}
```

Figure 2.5: Code snippet of user transaction

User Transaction

Additionally to the CMT and BMT schemes, both controlling transactions at the AS, the J2EE AS also provides an approach called *User Transaction* to control transactions from the client side. Using the user transaction object, the client program can explicitly mark the boundaries of a transaction. Figure 2.5 provides a sample code showing how a client program uses the user transaction to associate client requests with transactions. The begin method of the user transaction object will trigger the TM of the AS to start a new transaction. Within the transaction, the client can send one or more requests to one or more EJB objects. Finally, the commit/abort method of the user transaction object lets the TM commit/abort the transaction.

Execution patterns

In a J2EE AS, CMT, BMT and user transactions can be used independently or together within a single application. Thus, different ways to use these approaches can lead to a variety of associations between client requests and transactions, i.e., to many different execution patterns. According to our analysis shown later, we find that the variety will cause different side effects when a crash occurs, and thus affect the design of the replication algorithm.

Although there could be many different execution patterns, only some of them make sense in practice. Hence, this dissertation focuses on those patterns that are usually applied in practical applications. In fact, the 1-1 pattern is the most common pattern, that can be implemented very easily using the default configuration of the CMT scheme. In the CMT scheme, the default transaction

attribute of each EJB method is `Required`. Hence, when a client request is submitted to the EJB object, a new transaction is created for the execution of the client request since yet no transaction is associated with the execution thread. If during the execution, sub-requests are submitted to other methods of the EJB, all these executions will be within the same transaction since the transaction is already associated with the execution thread. This way, the association between the client request and the transaction is the 1-1 pattern. Chapter 4 will discuss in detail not only the 1-1 pattern, but also other patterns, and show how they can occur in J2EE through a combination of CMT, BMT, and user transactions.

Transaction Abort

When a global transaction is aborted, the full state consistency requires the AS to rollback state changes performed by the corresponding AS transaction. In the BMT scheme, the rollback operation can be included in the code of the EJB method as shown in Figure 2.4. However, in the CMT scheme, the case is different, since the abort operation is not explicitly controlled by the transaction interceptor. J2EE addresses this issue by providing a `SessionSynchronization` interface to let an EJB instance be notified of the boundaries of a transaction by the container. The interface has a method called `afterCompletion(boolean committed)`. An EJB class that implements the `SessionSynchronization` interface has to implement the `afterCompletion` method. The rollback operations can be implemented in this method under the condition that `committed` is false. Then, when a transaction that accessed an EJB instance of this class is aborted, the TM calls the `afterCompletion` method to do rollback operations. Thus, state changes performed by the transaction on the object can be automatically aborted, and full state consistency is guaranteed. The rollback action implemented in the `afterCompletion` method should not access the database since the DB transaction has aborted. They should also not access other components to prevent disseminating the abort to other objects. If more than one object is involved in an aborted transaction, each of them runs its own `afterCompletion` method without interfering with each other.

2.5 Overview of Failures

A *failure* of a system means the observation of deviation of the system from its specification [86]. Before a failure is observed, the deviation is called *error*, i.e., an abnormal state of the system. Error is caused by some defect in the system. Almost all systems have defects, and hence error is almost inevitable for any system. The goal of fault-tolerance is to prevent deviation being observed when an error occurs in the system, i.e., to make the observed behavior of the system look like a non-faulty system. This is also the base line to evaluate the correctness of a fault-tolerant algorithm.

2.5.1 Failure Types

A system might have different kinds of failures. Different failures have different side effects, and the corresponding fault-tolerance algorithm has to address the differences. Failures can be classified as following [28]:

- A *crash* failure occurs when the system stops working completely. If the clients of the system can eventually detect the failure, the failure is called fail stop. Otherwise, it is called fail silent. The typical reasons for a crash can be categorized into: (1) programming error (e.g., deadlock, or stack overflow), (2) OS error (e.g., OS crash), and (3) catastrophic error (e.g., power cut).
- An *omission* failure occurs when the system does not respond to a request when it is expected to do so. When the omission failure takes place, the system might still keep working. A typical reason for a omission failure is a network partition between the client and the system.
- A *timing* failure can occur in real time systems if the system fails to respond within the specified time slice. Both early and late response might be considered as timing failures. Late timing failures are typically caused by some bottleneck in the system or in the network.
- A *Byzantine* failure occurs if the failure makes the system behave arbitrarily.

This thesis focuses on crash failure and assumes no omission and Byzantine failures occur. We assume reliable, asynchronous communication and no network partitions because we believe that assuming no network partitions is reasonable for a LAN environment. Timing failures are not considered because we do not look at timing requirements. With this, if a non-replicated system does not crash, clients of the system eventually receive correct responses for all requests. Furthermore,

we do not consider programming errors. How to tolerate programming error is an important topic, but it is out of the scope of our research.

2.5.2 Crash of an Application Server

When an AS crashes, we think it is not accessible any longer, and all connections between the crashed AS and its clients and the backend database are lost. The crashed AS might be recovered later as a new instance. Before designing a correct replication algorithm to tolerate crash, it is very important to understand the side effects of a crash of a non-replicated AS.

i. The crash causes the crashed AS to lose its state, since the state is volatile in memory. Some AS systems have built-in persistence mechanisms that can make its state persistent locally. However, as mentioned before, the persistent state is not accessible until the system is restarted. If the AS system logs its state in a database or a file on another machine and the persistent state can survive crash, we consider this logging mechanism as a special form of replication (please see Section 2.6.1).

ii. For any request executing on the AS system, if the crash takes place before the response of the request is returned, the crash causes the client not to receive the normal response. Instead, in practice, the client usually receives an exception to show the disconnection to the server or time out. That means, the client can detect the crash. Hence, in this thesis, we assume that the crash failure is fail stop. Our solution achieves that the crash exception is invisible to the client program to guarantee transparent fault tolerance.

iii. The crash can also affect databases that were called by the crashed AS. Assume an AS transaction had submitted a set of requests to a database before it crashed. On the database, the corresponding DB transaction is still active at the time of crash. According to the specification of transaction services in J2EE and CORBA [76], the AS transaction has one connection to the database, and the association of the AS transaction and the DB transaction is through this connection. When the AS crashes, the connection to the database breaks, and the DB transaction is aborted. The abort undoes all state changes done so far on the database. In summary, when an AS crashes all active DB transactions that had a corresponding AS transaction at the AS are

aborted.

One of the worst cases is that the AS crashes in the middle of the 2PC of a transaction, where some databases have returned the *prepared* message but have not received the *commit/abort* decision yet. In this case, these databases will keep the state changes made by the transaction, keep locks on those changed data, and always wait for the final decision. Nevertheless, as DBMS are the most prevalent backend tier, we only consider standard DBMS behaviors. However, if the system called by the crashed AS is not a database, or the request to the database is executed according to some other specification, the callee system might have a different reaction, e.g., stop where it is, continue to execute the request, etc.. In this thesis, we assume that the database will undo all state changes made by the transactions that are active at the time of crash except those in the *prepared* state (in case of 2PC) which will remain active until the database receives a *commit/abort* decision.

2.6 Tolerate Failures through Replication

Replication is an efficient mechanism to tolerate crash failure. Replication can be *active* or *passive* [104]. In the active scheme [89, 7, 35, 72], a request is sent to and executed at all replicas. The crash of a replica will not affect execution on other replicas. The client receives a response as long as one replica is available (duplicate suppression must be in place). In passive replication [22, 51, 44, 73, 41, 11, 62], only the primary replica executes the request, and propagates updated state to the backup replicas. If the primary fails, failover takes place, and one of the backups becomes the new primary, installs the up-to-date sate, and continues working. Requests that are active on the primary at the time of crash should reexecute on the new primary. The new primary has different ways to know which requests are required to be reexecuted. A first option is that reexecution relies on resubmission after crash. In this way, after the crash, clients resubmits all active requests whose responses were not returned before the crash, to the new primary. This is the typical way used in most commercial products, such as JBoss [60] and WebLogic [14]. A second option is that reexecution relies on the multicast of requests. This way, each client request is multicast to all replicas during normal processing and is recorded at each backup. Thus, after failover, the new

primary already knows the requests required to be reexecuted. The Eternal system [73] uses this way to do passive replication. A third option is that reexecution relies on a request log. Each client request is logged during normal processing. After failover, the new primary reexecutes interrupted requests by reading them from the log. The Phoenix system [9] uses this option. For replayed requests that are interrupted by the crash, the resubmission mechanism has a longer delay than the other two, since requests have to be resubmitted by the client tier. Whereas, it saves time during normal processing since it does not require additional time to multicast or log requests. To get better performance during normal processing, we use the resubmission mechanism in our solution.

Active replication requires deterministic behavior (otherwise, complex consensus mechanisms such as are required), and induces heavy load, since all replicas have to execute all requests. A new approach called Midas, proposed in [92], lets active replication live with non-deterministic behaviors by running compensation code that is generated by statically analyzing application source code to eliminate inconsistent states caused by non-determinism. Passive replication allows for non-determinism. Although primarily designed for fault-tolerance, it has some potential for scalability, since applying changes sent from the primary is usually less time consuming than executing the requests themselves. The spared resources can be used to perform other tasks. Furthermore, in our solution, each replica is a primary for a subset of requests, and backup for the others. On the negative site, passive replication requires complex state propagation and failover. There also exist some research considering a combination of active and passive scheme (e.g., [31, 32]) and some replication tools support both active and passive replication (e.g., [74, 114]).

2.6.1 Passive Replication Category

When looking at commercial AS products, almost all rely on passive replication (e.g., Phoenix/.Net [9], WebLogic [14], WebSphere [103], JBoss [49], Sun AS [96], Oracle9i [78], IONA E2A [56], Pramati [85], Orion AS [57]). Passive replication can be categorized by two parameters:

i. The primary can replicate state changes to the backups in different ways. Using *cold replication*, the primary stores the state information on a persistent storage which can be accessed at failover by the new primary. This mechanism is also known as logging or checkpointing. In this case, the new primary needs only to be initiated when needed after a crash. Using *warm*

	Eager	Lazy
Warm	JBoss/Oracle9i	WebLogic
Cold	IONA E2A/Pramati/Sun AS/Phoenix	Orion AS/Phoenix

Table 2.1: Classification of passive replication of commercial products

replication, the primary sends state changes to the backups directly, e.g., via messages. This alleviates the load on the persistent storage but introduces message overhead. Backup instances must exist to receive replicated state. During normal processing, the time to do replication depends on the communication time between the primary and backups for warm replication, and depends on the communication time between the primary and the persistent storage for cold replication. In Section 10.1.1, the performance evaluation shows that the replication time of warm replication is faster than that of cold replication. Moreover, failover of warm replication is also faster than that of cold replication, since in warm replication the backup has already the state in memory, while in cold replication it needs to read the replicated state from the persistent storage first.

ii. The *propagation time* defines when state propagation takes place. The propagation time is demarcated by the boundary of transactions. If state changes are propagated before the related transaction commits, we say it is *eager* propagation. Otherwise, if state changes are replicated at some time after, we say it is *lazy* propagation. When using eager replication, at the time a DB transaction commits, all AS replicas have the state changes of the associated AS transaction. This makes it possible to guarantee in case of primary crash that the state of the new AS primary and the DB is consistent. In contrast, in lazy replication, the AS might lose state changes as the old primary might crash before propagating changes of an AS transaction for which the corresponding DB transaction has committed. As a tradeoff, eager replication increases user response time by adding the time to do synchronous replication, but lazy replication provides fast response time.

Table 2.1 shows a classification of the schemes used by commercial products. Most of them use eager replication to guarantee consistency. Phoenix uses a lazy approach for deterministic requests, but uses an eager approach for non-deterministic requests. The products using warm replication

usually initiate a server cluster at system start-up and then choose one as the primary. The products using cold replication usually only start a single server and start the new primary only after the old primary crashes. As one of the main focuses of this thesis are consistency and correctness, we use eager replication. Moreover, our experiments show that warm replication has better performance than lazy replication. Hence, we pay more attention to warm replication.

2.6.2 Correctness of Replication

In regard to correctness, an ideal replication algorithm should make the replicated system behave in the failure case in the same way as a non-replicated system behaves when no failure occurs. Informally, it means exactly once execution of each client request and consistent state changes at the AS and the database. On the client side, exactly-once execution means each client request receives only one response that is not a crash exception. This guarantees transparent fault tolerance from the client viewpoint. On the AS, exactly-once execution means each client request changes the state of the AS and/or the database exactly once. Consistent state changes on the AS and the database means correct and is "all or nothing" on both AS and the database in case of full state consistency or "all or nothing" on the database in case of relaxed state consistency. A replication algorithm is required to provide this form of correctness for a replicated AS even in the event of failures. We will discuss in more detail and more formally what correctness means in Chapter 4.

2.7 Communication Mechanism for Replication

When using replication to tolerate failures, failure detection and message propagation among replicas (for warm replication) are two important issues. How to detect failures in a distributed environment, and how to deal with possible message loss during communication are non-trivial problems. Fortunately, *Group Communication Systems* (GCS) [52, 26, 102, 39, 21, 19, 25] provide us powerful functionality. Examples of group communication systems include Spread [1], JGroups [50], ISIS [58], Horus [102], Ensemble [38], Transis [34], and Totem [71].

Figure 2.6 depicts the basic architecture of a group communication system. A set of applications build the group. Each application is a member of the group. When an application *sends* a message

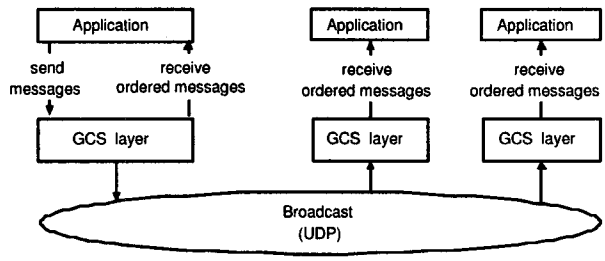


Figure 2.6: Group communication system

to all members, it first sends the message to the underlying GCS layer. The layer typically uses consecutive point-to-point communication or physical broadcast like UDP to send the message to all machines that have members of the group. At each site, the GCS layer first receives the message, then *delivers* the message to the member applications, at which time the application *receives* the message. When a GCS layer receives a set of messages, they may not be in the correct order. Thus, the GCS layer will reorder these messages before delivering them to the application. In this dissertation, according to the requirement of the algorithm, a set of AS replicas compose such a group, where each replica is a group member and uses GCS to manage the communication among all replicas in the same group. An AS replica can join one or more groups. Typically, a GCS provides two primitives: *group membership maintenance* and *multicast.*

2.7.1 Group Membership Maintenance

The membership service maintains a listing of the currently active and connected members and handles group operations such as joining and leaving a group. The output of the membership is called a *view.* At each site, the GCS contains a view V that contains the list of members with which communication is possible. An application might join or leave a group. This changes the view of the group. When a view change takes place, the GCS delivers a view change message to all members of the new view indicating that the new view is V'. The typical property for group membership is *virtual synchrony* [20, 26]: If members p and q receive the same new view V' while having the same previous view V, any message delivered to p, which is a member of V, is also delivered to q in V. This protocol guarantees that the GCS delivers exactly the same messages at all non-failed

members of the view V. Hence, the applications on the different sites perceive view change events at the same virtual time.

Since it is well known that accurately detecting failure in asynchronous environments is impossible [27], the membership service often uses an *inaccurate failure detector*, typically based on timeout: when the GCS does not receive any message from a member p beyond a time limitation, the GCS suspects p to be faulty, then excludes p from the current view [68]. In this case, p might have really crashed, or be partitioned from the network, or just be delayed by a slow connection. If the GCS excludes a correct member, we require that the affected replica shuts down itself and attempts to rejoin the group.

2.7.2 Multicast

The multicast sends a message from a member to all members in the current view. A GCS provides various multicast primitives with various degrees of reliability and ordering [52]. The possible reliability semantics are:

unreliable delivery no guarantee that a message will be delivered at all members,

reliable delivery when a message is delivered to member p, and if p dose not fail for sufficiently long time, the message will be delivered to all other members of the current view unless they fail,

uniform reliable delivery when a message is delivered to a member p, even if p fails immediately after the delivery, the message will be delivered to all other members unless they fail.

The difference between the uniform reliable delivery and the reliable delivery is on messages that might be delivered at failed members. With uniform reliable delivery, whenever a message is delivered at any member, no matter whether the member fails or remains available, all other non-failed member will receive the same message. Hence, the set of messages delivered to a failed member is a subset of messages delivered to surviving members. In contrast, with reliable delivery, a message might be delivered to a member that fails immediately after that, but the message is not delivered to other non-failed members. As a result, with uniform reliable delivery, when a member receives

a message sent by itself, it is guaranteed that all other non-failed members also receives the same message. Whereas, with reliable delivery, the sender of a message cannot know whether the message is delivered to other members by just testing if it has received the message itself. We will use uniform reliable delivery when the replication algorithm should not proceed before it is assured that all non-faulty AS replicas will receive a state change message.

The ordering primitive is very important when different messages might depend on each other. The offered ordering semantics are:

FIFO ordering If a member sends a message m before it sends message m', all members in the view receive m before m',

Causal ordering If a member sends a message m' after it receives message m, all other members in the view receive m before m',

Total ordering If a member receives a message m before it receives m' in a view, all members in the view receive m before m'.

In our context we will use FIFO and total ordering. The FIFO ordering guarantees that all members receive messages sent by a member in sending order. The total ordering guarantees that all members receive all messages in the same order.

Chapter 3

Traditional Application Server Replication Solutions and Correctness Criteria

This chapter reviews existing AS replication solutions. Most of them do not consider the effects caused by different execution patterns and relaxed state consistency, and only provide solutions for the simplest 1-1 pattern. Although some of these solutions might also be applicable for a certain advanced execution pattern, they do not clearly discuss this option. Most solutions either consider fault-tolerance or load-balancing, but not both. This chapter also reviews traditional correctness criteria for replication. These criteria normally only consider one tier, and do not distinguish the effects of different execution patterns.

3.1 Overview of Existing Replication Solutions for Fault Tolerance

In this section, we review existing replication solutions for three prevalent AS specifications, namely J2EE, CORBA and .NET, and investigate which execution patterns are assumed for them.

3.1.1 Replication for J2EE Architecture

Most J2EE products use passive replication based on the resubmission mechanism. A typical example is JBoss's clustering solution [60]. It uses passive, warm, and eager replication. Each replica can act as a primary for a client session. If a client request triggers execution on several stateful components, state transfer takes place individually for each component once execution on this component has terminated. Problems occur if replication on some components was successful but the primary crashes before the corresponding database transaction commits. In this case the backups have a partially replicated state while the database transaction aborted. Obviously, the state of the new primary is inconsistent with the state of the database. Hence, this replication solution does not work correctly even for the 1-1 pattern.

Some commercial products, like Oracle 9i [78] and IONA E2A [56], do replication at the end of each client request. Pasin et al. [83] propose a High-Available EJB server architecture where the state changes are replicated at commit time. For the 1-1 pattern, these mechanisms are similar to ours since the end of each client request is the commit time of the transaction associated with the client request. However, there always exists a time difference between the time to do state propagation and the time to do commit. Thus, like the JBoss solution, state inconsistency occurs if the primary crashes after state has been successfully propagated but before the corresponding database transaction commits. In this case the backups have the state changes while the database transaction aborted. Although there exist some mechanisms to coordinate between the AS and the DB, they are not clearly described. Moreover, above solutions do not consider advanced patterns. Kistijantoro et al. [62] also propose to do replication at time of commit. The solution checkpoints the state changes on AS into the database within the context of the transaction, and hence the state changes on the AS and on the database are consistent. As a result, this solution works for the 1-1 pattern. Pramati [85] uses a similar solution to persist states at the time of commit and hence guarantees consistency for the 1-1 pattern. However, both solutions do not consider advanced patterns and relaxed state consistency.

One of the leading products of AS, WebLogic [14], uses passive, warm, and lazy replication. Each EJB instance has a single primary server, which processes requests to the EJB instance, and

propagates the state of the instance soon after returning the response to keep replicas as consistent as possible. Due to lazy replication, inconsistencies arise when the primary crashes after the database has committed a transaction, but before the corresponding state changes on the AS are replicated. Orion AS [57], which also uses lazy replication, has a similar problem. Another famous AS product, WebSphere [103], does not replicate the state of the AS, and hence, its clustering only works for stateless applications.

Existing replication solutions for J2EE AS products usually have their main emphasis on ease of implementation, as they are mostly industry solutions. Most of them do not change the interfaces between AS and clients, and between AS and database, and do not require any additional support from other tiers. As a tradeoff, these solutions sacrifice consistency. Until now, we are not aware of any commercial J2EE product, that clearly provides even for the 1-1 pattern a replication solution that guarantees full state consistency.

3.1.2 Replication for CORBA Architecture

Although there are less AS products based on CORBA than on J2EE, there exists more research on replication in CORBA than in J2EE. While most existing J2EE solutions are quite simple, many CORBA solutions have an advanced framework supporting both active and passive replication, such as [29, 30, 72, 65, 40, 74, 7]. Some solutions combine active and passive replication as semi-passive solutions ([31, 32, 15]). These research projects consider the internal architecture of CORBA, and the replication solution is tightly bound to CORBA. For active replication, the solutions have to make sure that all replicas receive the same requests in the same order, e.g. by using the total order delivery of group communication systems ([30, 40, 47, 72, 42]). For instance, Marchetti et al. [69] propose to build a sequencer service based on the total order delivery between clients and server replicas to guarantee that all server replicas execute client requests in the same order while messages of requests might be arbitrarily delayed or timeout between clients and server replicas. For passive replication, either warm or cold replication is used to replicate state changes. When using warm replication, to guarantee that all replicas receive the same changes, some solutions use a group communication system to reliably broadcast state changes ([29, 74]). Other solutions use a 2PC protocol, where the backups are the participants of the 2PC ([65]).

In here we look at the Eternal system [73, 74] in more detail as an example in CORBA. Eternal is based on the FT CORBA architecture [75]. It supports both warm and cold passive replication. The primary replicates the state to backups periodically in form of checkpoints. Between two checkpoints, all messages from clients and the database are logged. At crash of the primary, the new primary first restores the state of the last checkpoint, and then replays logged requests. At recovery of a replica, the primary transfers the last checkpoint state to the recovering replica. Zhao et al. [114, 115] extend the Eternal system [74] so that CORBA components can access a backend database. A special replicated out-bound gateway is used to manage the transaction context between the application server and the database. Database connections are protected by the outbound gateway. A response from the database will be replicated to all backups. The replay mechanism assumes deterministic execution at the AS. If a new primary has to reexecute requests at failover time, duplicate database access can be avoided by directly taking the replicated response without reexecution. Although the solution does not look at different execution patterns, we think it can support them with some extensions. However, if non-determinism exists, the solution cannot guarantee correctness even for the 1-1 pattern, since reexecution might generate responses or database accesses that are different from the logged information. This will lead to executions that do not follow the original execution path. Moreover, the solution is not based on the common interface between the AS and the database. Instead, it depends on a special transaction manager that does not directly connect to the resources but multicasts the transaction requests to the out-bound gateway.

Two other solutions proposed by Felber and Narasimhan [41] and by Frølund and Guerraoui [43] use a much simpler marker mechanism to coordinate state changes on the AS and on the database. The former solution acts similar to the J2EE replication solution of [62, 85], checkpointing the state changes on the AS of a given AS transaction into the database within the context of the corresponding DB transaction. Then, at failover time, the new primary checks the database. If the transaction aborted, neither DB nor AS changes exist. Otherwise, the new primary can get the AS state changes from the database. The latter solution propagates the state changes of the AS to the backups immediately before the commit and then inserts a marker into the database as part of the DB transaction. At failover time, the new primary checks the marker for each transaction. If a maker exists in the database, it means that the database has already committed the DB transaction and has

the state changes related to the persistent data. Thus, the new primary installs the corresponding state changes of the AS components. Otherwise, the database had aborted the DB transaction and the new primary discards the AS changes. Although the details of these two solutions are different, both solutions are taking advantage of the transaction's property to coordinate the state changes of the AS and the database. This mechanism works well for the 1-1 pattern and does not require additional support from the database. We use the same idea in our algorithm. However, these two solutions do not consider advanced execution patterns and relaxed state consistency. In fact, although there are many CORBA-based replication solutions, only a few of them regard a CORBA-based AS as the middle tier of a multi-tier system. Furthermore, although several solutions can correctly handle the 1-1 pattern, so far, none can correctly handle advanced patterns and relaxed state consistency.

3.1.3 Replication for .NET Architecture

For Microsoft's AS platform .NET, the main replication solution has been developed in the Phoenix project ([11, 9, 10, 8]). It has similarities to the Eternal system. State is replicated periodically, and requests between two checkpoints are logged. Failover starts from the last checkpoint and applies logged requests assuming piecewise deterministic behavior [37]. It requires the database to be able to identify duplicate requests and log replies. This would be possible, if a persistent queue exists between AS and database. Unlike the Eternal system, it distinguishes non-deterministic events from deterministic events. For non-deterministic events, it uses eager replication (namely immediately logging the result of these events before returning) to guarantee consistency. Although the papers present a formal discussion of correctness, the transactional properties are not clear. Although eagerly replicating results of non-deterministic events enables the algorithm to support non-determinism in some cases, it is not sufficient. A problem is the database access. When the primary crashes, active transactions will abort at the database. Then, during reexecution of these transactions, the replayed database accesses might get logged replies without real reexecution on the database. As a result the database might miss the state changes of these replayed transactions.

3.2 Traditional Correctness Criteria for Replication

Traditional correctness criteria normally only focus on one aspect or on one tier, ignoring the global picture of the entire system. These criteria often dig into depth into one aspect, e.g., concurrent data access, or the ordering of messages, but do not consider the relationship between different tiers and different execution patterns.

In this section, we look at three well-known traditional correctness criteria for replication mechanisms: one-copy-serializability, state machine replication, and X-ability.

3.2.1 One Copy Serializability

One copy serializability (1CSR) [16] has been developed for replicated databases. It addresses correctness in regard to two aspects: multiple copies of a data object must appear as a single logical copy (1-copy-equivalence) and the effect of the concurrent execution of transactions must be equivalent to a serial execution (serializability).

To achieve 1-copy-equivalence, read and write operations on logical data items have to be translated to serial operations on the physical data copies. When using eager replication, a simple approach to do so is *read-one/write-all* (ROWA) [17, 16], which requires write operations to access all copies while read operations are done locally at one copy. Alternatively, *quorum protocols* [101, 48, 59, 82] require both read and write operations to access a quorum of copies. As long as a quorum of copies agrees on executing the operation, the operation can succeed. When using lazy replication together with primary copy, 1-copy-equivalence can be guaranteed only in the primary copy, backup copies are only ensured to be eventually equivalent.

To guarantee serializability, concurrency control mechanisms are required. A typical example is locking, e.g., 2-phase-locking (2PL). If updates are always first executed on a primary replica, local concurrency control on the primary is enough. Whereas, if updates can be done concurrently on different copies, distributed locking is required. When proving a replication algorithm provides 1CSR, one has to show that all executions that are possible under the given concurrency control and propagation approaches, are equivalent to some serial execution on a single logical copy of the database. For passive replication with a single primary replica, 1CSR is not difficult to achieve,

since local concurrency control on the primary is sufficient to guarantee serializability, and eager replication can easily guarantee 1-copy-equivalence.

However 1CSR has been designed for database replication, and thus is not appropriate to check correctness for AS replication, since it does not consider the interaction between the replicated AS and a backend tier. For instance, it cannot check duplicate or missed requests to the database.

Linearizability [53, 5] is another well-known criteria to evaluate correctness of concurrent executions. While serializability is addressing the internal sequence of concurrent transactions on a shared object, linearizability considers the external observed sequence of concurrent operations on a shared object. It does not address transactions. Instead, it considers the sequence of requests and responses. For instance, given two concurrent requests r_1 and r_2 on the same object, if r_1's response is perceived before r_2 is invoked, the linearizable execution sequence of r_1 and r_2 must guarantee that r_1's execution is before r_2's execution. Although linearizability could be a useful criteria for the replicated AS to check if the ordering of requests and corresponding responses is preserved linearly in case of resubmission, it is not a sufficient correctness criteria for replicated AS yet, since it does not consider the interaction between the replicated AS and a backend tier.

3.2.2 State Machine Replication

State machine replication [88] is a very well-known formalism for active replication. The replicated system is modeled as a state machine, and each replica has a replica of the state machine. A request triggers actions on all state machine replicas to transfer the state machine from the same initial state to the same final state. The correctness criteria addressed by the state machine replication is that all replicas receive and process the same sequence of requests. It has two requirements: (1) every correct replica receives every request (Agreement), and (2) every correct replica processes requests it receives in the same relative order (Order). Then, based on the assumption of determinism, all replicas executing the same sequence of requests based on the same initial state will reach the same final state. Using a group communication system to multicast requests to all replicas can easily fulfill the agreement requirement via uniform reliable delivery, and the order requirement mechanism via the total ordering mechanism. For example, in [112], the state machine replication is used to build a fault tolerance framework for web services and the total ordering of requests is guaranteed by a

consensus-based algorithm.

However, when considering that an execution might submit sub-requests to other tiers, e.g., the database, more problems arise. For example, connections to the database are a problem. Typically, there are three possibilities. The first option is that each replica has an individual connection to the database, and the database needs to detect and suppress duplicate requests and needs to guarantee that the crash of a replica will not affect connections from other replicas. The second option is that only one replica builds a connection to the database, and the replica needs to pass responses to all other replicas afterwards. If the replica crashes, the original connection is lost, and another replica must be chosen to connect to the database. In this case, problems about missed requests and duplicate requests might arise. The third option is that all replicas connect to a specific gateway, and the gateway filters duplicate requests and builds a connection to the database; however the gateway itself might need to tolerate failures using replication, and hence, similar problems arise again. The Eternal system [73, 74] uses the gateway approach. All of these problems might arise even for deterministic execution, but so far we are not aware of any correctness criteria based on state machine replication that would address them.

As an alternative to group communication, consensus mechanisms [25, 31] are widely discussed for active replication to guarantee all replicas agree on the sub-requests and the responses despite non-determinism or Byzantine failures. However, consensus mechanisms do not consider issues such as duplicate requests, missed requests, and state consistency between tiers. Hence, we still need a correctness criteria to consider these problems.

3.2.3 X-ability

The X-ability framework [44] allows reasoning about correctness in a multi-tier replicated system based on execution histories. It takes into account that the replicated tier can call other tier in the system. X-ability assumes that in the non-faulty non-replicated case, many different execution sequences are possible for a given set of requests. Such a sequence is called a failure free execution history. If the tier is replicated and some replicas might crash, the execution sequences for these requests will be more complex, since some requests will be interrupted by the crash and reexecute on other replicas. A sequence in a replicated system with failures is called a real execution history.

X-ability proves that a replication algorithm is correct for a tier by checking if all real execution histories on the tier, which are possible under the given replication algorithm, can be reduced to failure free histories. The rule of reduction is that if an interrupted execution of a request and the corresponding reexecution have the same side-effect as that of a possible failure free execution of the request, then the interrupted execution and the reexecution can be reduced to the failure free execution. X-ability considers an execution to possibly change the state and invoke servers of other tiers. This reduction mechanism implicitly checks missed requests and duplicate requests for the replicated tier. X-ability assumes a request to another tier is either idempotent or undoable. If a request is idempotent, no matter how many times the request reexecutes, the sum of these executions has the same effect as the idempotent failure free single execution. If a request is undoable, before the request reexecutes, the side effect of its last execution can be undone, and hence the last successful reexecution can be considered as a failure free execution.

X-ability allows reasoning about the correctness of a composite system very easily by assuming that each tier provides X-ability. As such, proof of correctness can be done independently for each tier. For instance, at failover, a replication algorithm for a tier might restart any execution that was active on a crashed replica at time of failover. If the original execution on the crashed replica had submitted a sub-request to another tier, the reexecution might resubmit the very same request. This, however, is not problematic since such request is assumed to be idempotent, hence, a resubmission does not lead to any inconsistency. However, things are more complicated if a called component does not provide X-ability. For instance, a backend database system usually does not provide X-ability, since it does not provide idempotent operations or the possibility to undo committed transactions. A replication algorithm for an AS has to take this into account. In this case, proof of correctness cannot be done independently on the replicated AS; both the client tier and the database tier are required to be considered. In this thesis, we model execution that goes beyond an individual tier in order to reason about such cases.

3.3 Load Balancing and Combined Approaches

Load balancing and fault-tolerance have traditionally been handled as orthogonal issues, and research on one topic usually does not attempt to solve the other. Section 3.1 has pointed out that the primary-backup approach is used in many AS replication solutions for fault tolerance purposes. The main differences between these solutions are when to do replication and whether or how to guarantee state consistency after failover. Most solutions we discussed in Section 3.1 only use the primary replica to execute all the load, and do not aim at scalability.

However, for load balancing purposes, we need more replicas to be able to execute requests to share the load. Typical load balancing solutions of application servers (or web servers) use a centralized load balancer (also called scheduler) to manage to distribute the load to different replicas. In content-blind policies [4], such as Random or Round Robin, the load balancer does not know the load on each site. As content-blind policies can be easily implemented, they are widely used in practice. However, they do not work well in heterogeneous environments. Content-aware policies require some knowledge about the environment. There exist many strategies, e.g. sending requests to the least loaded replica [87, 79], distributing requests according to data size [111], or locality of requests [81, 3, 36]. Feedback-control and resource consumption predictions are other mechanisms [110, 66]. Such strategies can dispatch load more precisely, but either need a central scheduler with global knowledge or require frequent exchange of load information. Central schedulers present a single point of failure. Replicating them is possible but has its own overhead. In contrast, our content-blind approach with request forwarding is purely distributed with little overhead, and is easy to implement.

Several commercial solutions (e.g., used in JBoss [60], Weblogic [14] and Sun Application Server [96]) use component replication for both fault-tolerance and load-distribution. In the basic approach, all components are replicated on all servers and requests are balanced across all replicas in the clusters. If a replica fails, any replica can take over. However, as was discussed in Section 3.1, many of the commercial replication solutions do not work correctly in the presence of failure. Furthermore, the approach does not provide enough scalability since replicas spend too much time on

backup activity when the cluster size increases. The scalability problem can be overcome by partitioning a cluster into sub-clusters and let a component be only deployed and replicated in a sub-cluster. Therefore, updates only need to be propagated in the sub-cluster leading to less overhead. However, requests to this component can also only be distributed to this sub-cluster. Furthermore, the approach requires to artificially define a sub-cluster for each component which makes reconfiguration complex. In our approach, a component is deployed on all replicas. However, at run time, component instances are only replicated on a fixed number of replicas which is independent of the size of the entire cluster.

Only a few approaches in the research literature consider both load distribution and fault-tolerance. Singh et al [91] propose a system that merges the Eternal [74] fault tolerance architecture and the TAO's load balancer [87]. All servers in a cluster are partitioned to several disjoint FTG groups. A similar architecture is used in [80]. However, only the primary server in each replica group is used for load balancing while backups do not contribute to load distribution. Moreover, these solutions do not address reconfiguration problems. Long et al [67] propose a solution in which each server acts as both a primary and a backup. However, their solution is specific for a cluster with only 2 machines. In [84] all components are replicated on all replicas leading to limited scalability. Other combined solutions do not consider stateful AS. For example, Ho and Leong [54] propose to replicate event channels and share the load among replicas using the replicated channel. However the approach only replicates stateless event channels.

Chapter 4

Execution Patterns for Application Server

In Section 2.6.2, we discussed informally what it means for a replication algorithm to show correct behavior. Informally, the execution despite possible failures of individual components should be equivalent to the execution in a non-faulty non-replicated system. The challenge in defining correctness lies in the many different ways client requests and their execution at the AS and database tiers can be coupled with the notion of transactions that require atomicity, durability and isolation.

This chapter addresses this issue by formally modeling a set of execution patterns that reflect the most common way in which client requests and transactions are associated with each other in a 3-tier architecture. In order to do so, we first model request executions. Then, transaction execution is modeled using the concept of execution patterns. We then model state changes performed by transactions. At last, we derive a set of correctness properties for the execution in a non-faulty non-replicated 3-tier system.

The model is then extended to include the behavior in case of a crash and typical actions of a passive replication algorithm. Based on this extended model, a set of correctness criteria is presented that a replicated system should fulfill in order to emulate a non-faulty non-replicated system.

4.1 Request Execution

We refer to *Requests* as the set of all requests. *Actions* is the set of all possible actions triggered by requests at the AS and database and *Responses* is the set of all possible responses. We assume each client C_i first establishes a session with the AS and all requests of this client are executed within this session. We denote the session of the client C_i from the first request establishing the connection to the last response before disconnection as a special *Action* CA_i. Furthermore, we assume that each client submits requests in a single thread based on the blocking scheme. That is, a client only submits the next request when it has received a response to the previous request. Each $r \in Requests$ submitted to the AS by CA_i triggers an $a \in Actions$ on the AS. We refer to this as an *AS action*. The action performs read and write operations on the state of the AS and performs calculations. Please note, we assume that the client never accesses the database directly.

Additionally, an AS action a might make further calls to the database or to other components on the AS again based on the blocking scheme. A call to the database is typically an SQL statement. We refer to this as a request $r \in Requests$ and it triggers an action $a \in Actions$ on the database referred to as a *DB action*, which executes the SQL statement. This includes read and write operations on the data of the DB. After the completion of an AS or a DB action, a response $rp \in Responses$ is returned to the caller. When an AS action a makes calls within the AS, in some cases, the calls are also requests, triggering new (nested) actions on the AS. Sometimes, the execution of the call is considered part of the action a. We will discuss this later, when we introduce transactions.

Thus, an action refers to a set of operations on one tier. It has a unique corresponding request and one response. The function $R(a)$ represents the request leading to the action a, and $RP(a)$ represents the response provided by a. The signature function $SIGRP(rp)$ indicates the action that returned the response rp (i.e., $a = SIGRP(RP(a))$. Due to non-determinism, a request might cause different actions (different set of read and write operations), depending, for example, on the previous state of the AS/DB. The function $A(r)$ represents the set of actions that might be triggered by the request.

Figure 4.1 denotes a sample execution of all requests within the client session CA_i. In the figure, the first line represents time at the client, the second represents time at the AS, and the

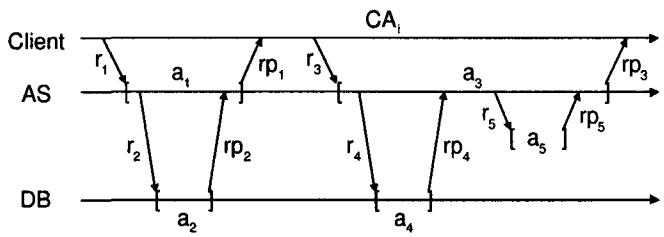


Figure 4.1: Sample scenario of request execution

third represents time at the database. The boundaries of an action are denoted with [and]. In the execution, request r_k triggers action a_k which returns response rp_k. The client submits requests r_1 and r_3. AS action a_1 submits request r_2 to the database triggering action a_2. AS action a_3 submits request r_4 to the database, triggering DB action a_4, and then submits request r_5 within the AS, triggering the nested AS action a_5.

4.1.1 Histories of Request Execution

As execution is single-threaded, we can specify a strict ordering of requests, responses, and actions related to one client. We denote as the *request history* RH_a of action a the sequence of requests submitted by a. For instance, in Figure 4.1, $RH_{CA_i} = r_1 r_3$, and $RH_{a_3} = r_4 r_5$. Similarly, we denote as *response history* RPH_a of action a the sequence of responses that a receives for the requests it submits. We denote the number of requests in the request history RH_a with $|RH_a|$ and the position of a request r in the history RH_a with $POS^r_{RH_a}$. Similar notation $|RPH_a|$ and $POS^{rp}_{RPH_a}$ is used for the cardinality of a response history RPH_a and the position of a response RP in the history RPH_a. Due to our model of blocking calls, it is always true that either $|RH_a| = |RPH_a|$ or $|RH_a| = |RPH_a| + 1$.

An important property of a correct execution is that the request and response histories must match.

Definition 4.1.1. *Let a be an action. We say RH_a and RPH_a match (denoted as $RH_a \bowtie RPH_a$) if the following holds:*

1. $\forall r \in RH_a$: *eventually* $\exists rp \in RPH_a, POS^r_{RH_a} = POS^{rp}_{RPH_a}$.

2. $\forall rp \in RPH_a$: $\exists r \in RH_a, POS^r_{RH_a} = POS^{rp}_{RPH_a} \wedge SIGRP(rp) \in A(r)$.

The above indicates that each request in the request history triggers one of the possible actions for this request and this action returns the appropriate response. For a client action CA_i, the matching is denoted as $RH_{CA_i} \bowtie RPH_{CA_i}$.

4.2 Transactions and Execution Patterns

Per our assumption in Section 2.4, all execution (read and write operations on data) have to be performed in the context of transactions. As outlined before, transactions can be triggered in various ways – either implicitly by the container, explicitly by the client or explicitly by the application programmer within the code executed by the AS. We now derive a set of execution patterns, which are denoted as 1-1, N-1, 1-N, or N-N to indicate the number of client requests involved in a transaction and the number of transactions generated by a request.

Recall that we split a global transaction in an AS transaction, which is denoted as $AST(t)$, and a DB transaction, which is denoted as $DBT(t)$. Accordingly, for an AS or DB transaction t', $GTX(t')$ indicates its global transaction. For the global transaction t, both $AST(t)$ and $DBT(t)$ must terminate in the same way. Namely, if $DBT(t)$ commits (aborts), then $AST(t)$ has to commit (abort), and vise versa. Please note, every global transaction must have a corresponding AS transaction since transactions are always started at the AS. However, a global transaction might not contain a DB transaction since AS actions involved in the transaction might not access the database. Furthermore, for a transaction t, we denote the client of the transaction as $CL(t)$.

We now discuss our execution patterns in more detail. We first describe them informally, then discuss their use in practice and then define them formally.

4.2.1 1-1 Pattern

The *1-1 pattern* (1 request - 1 transaction) means the execution related to a single client request is encapsulated in one *global* transaction which spans operations at the AS and possibly operations

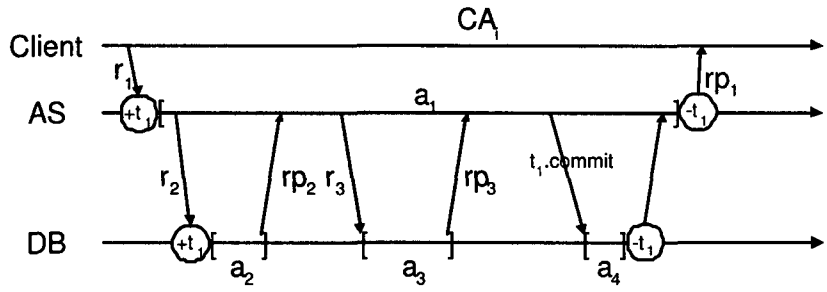


Figure 4.2: 1-1 pattern

at the database. As introduced in Section 2.4.4, it is the default execution pattern in J2EE when EJB objects use the CMT scheme. Therefore many applications use exclusively the 1-1 pattern. Figure 4.2 illustrates a sample 1-1 pattern. When the AS receives client request r_1, the TM starts a transaction t_1 on the AS. Then, r_1 is executed within t_1 as an AS action a_1. During a_1, a sub-request r_2 is submitted to access the database. At this moment, the DB transaction of t_1 is started at the database. The next sub-request r_3 is executed within the same DB transaction. At the end of the execution, the TM submits the commit request $t_1.commit$ to commit t_1's DB transaction. Then, t_1's AS transaction commits at the AS. At last, r_1's response rp_1 is returned the client. In the figure, we use $+t_1$ and $-t_1$ to indicate the begin and end of transaction t_1 at the AS and the database. As each request triggers exactly one transaction, there is a single AS transaction per client request that coincides with the boundaries of the AS action triggered by the request.

From the above example, we can find that in the 1-1 pattern an AS action can submit several requests to the database that are all executed within the same DB transaction. The DB transaction starts with the first action on the DB and terminates with the last. Typically, the last request to the database is the commit request. However, the AS action could also request an abort so that both AS and DB transaction abort. Furthermore, any database action might result in an abort (e.g., because of integrity violation) that then returns an abort notification as response to the AS action. In case of commit, the AS action and transaction usually terminate directly after receiving the commit response from the DB. In case of abort, an abort operation is required to rollback state changes performed by the transaction on the AS for full state consistency. Referring to Section 2.4.4, an

abort operation at the AS could be part of the original method execution in the BMT scheme, or be done by a specific abort method, e.g., executing the `afterCompletion` method in the CMT scheme. The response of an abort operation is an abort response. In both cases, we consider the abort activity as part of the original AS action. In case of an abort, the client typically receives a special abort response, which is denoted as rp_{ab_t}.

In summary, the 1-1 pattern has the following relationships. Each client request/response pair is associated with exactly one AS transaction. Thus, there is exactly one AS action per AS transaction. There is at most one DB transaction per AS transaction. Several request/response pairs between AS and DB can belong to this DB transaction. Thus, many DB actions can belong to one DB transaction. It also might be that a client request only triggers operations at the AS. In this case, there is only an AS transaction but no DB transaction.

4.2.2 N-1 Pattern

The *N-1 pattern* (N requests - 1 transaction) means several client/response pairs of a client are encapsulated within one global transaction. It is often used when a web-server (WS) runs between the real client and the AS. In this case, the real client makes a request to a component in the WS (e.g., a servlet) which makes in turn several calls to the AS. In order to guarantee all-or-nothing for the external client request, all calls to the AS should be embedded within the same transaction. In order to do so, the AS has to export the begin/commit/abort methods of the TM to the client. In an J2EE AS, it is the user transaction that enables the N-1 pattern. As shown in Figure 2.5, the client program can call the begin method of a user transaction object to start a transaction on the AS and call the commit or abort method of the user transaction to commit or abort the transaction on the AS. Between a begin and a termination request, several client requests can be submitted to one or more EJB objects. If each EJB method called within the transaction is in the CMT scheme and has the `Required` attribute, all these requests are executed within the same transaction. In this case, more than one client request is associated with a transaction. This behavior follows the N-1 pattern. Controlling transactions from outside the AS has also become important in the context of web-services.

Figure 4.3 illustrates a sample N-1 pattern. In this pattern, the client explicitly controls the

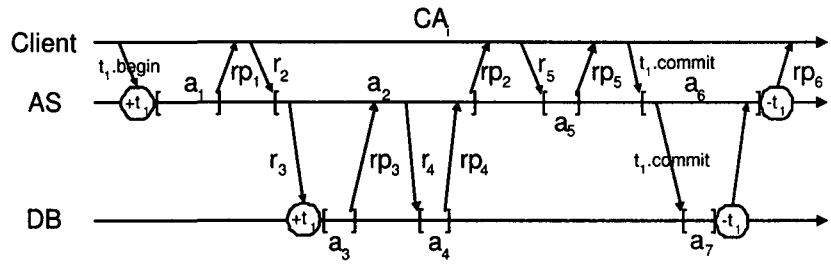


Figure 4.3: N-1 pattern

demarcation of transactions. The first client request starts the transaction on the AS, i.e., t_1 in the figure. Each following client request on behalf of t_1 triggers a new AS action. Each AS action can submit several requests to the DB, triggering several DB actions. Finally, the client submits a commit request $t_1.commit$ that commits t_1 at the DB and the AS. At any time an AS or DB action can trigger an abort, resulting in an abort of the entire global transaction that triggers an abort at the AS and the DB and an abort response to the client.

4.2.3 1-N Pattern

The *1-N pattern* (1 request - N transactions) means the execution of a single client request is associated with more than one transaction. Although this seems unusual at first, it is widely used in practice when a long execution needs to be chopped into small transactions in order to increase concurrency and decrease blocking within the database [90, 63]. It is then up to programmers to guarantee that the effect of executing a suite of transactions is the same as if there were only one big transaction. In particular, if not all of the transactions commit the effects of already committed transactions must be undone by executing corresponding compensating transactions provided by the programmer. Despite the added complexity, for applications where such compensation is easy, the advantage can be high.

In an J2EE AS, the 1-N pattern can be easily implemented in the CMT scheme using the `RequiresNew` attribute. When a client request calls a method of an EJB object configured as `Required` or `RequiresNew`, a transaction is started for the execution as in the 1-1 pattern. If

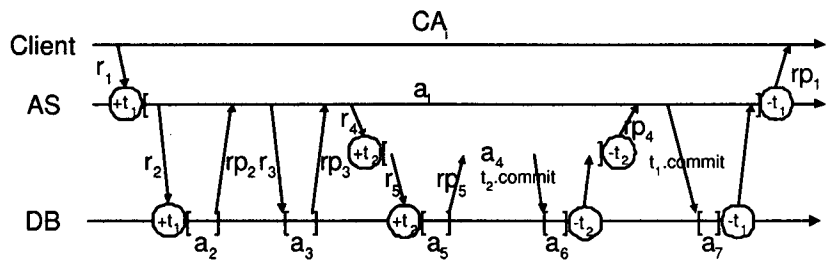


Figure 4.4: 1-N pattern

some EJB methods are configured as `RequiresNew`, when sub-requests are submitted to these methods during the execution of the client request, new transactions are created within the context of the existing transaction. In this case, the client request triggers more than one transactions at the AS. This association follows the 1-N pattern.

Figure 4.4 illustrates a sample 1-N pattern. Client request r_1 triggers transaction t_1 on the AS and then starts an AS action a_1 within t_1. t_1 starts and terminates with action a_1. t_1 starts on the database when a_1 submits the first sub-request r_2 to access the database. At the end of a_1, $t_1.commit$ is submitted to commit t_1 on the database and then commits t_1 on the AS. Transaction t_1 can have nested transactions, which are called *child transactions*, while t_1 is called a *parent transaction*. For instance, a_1 might make a call to an AS method that requires the start of a new transaction, e.g., calling a method marked as `RequiresNew`. This is the case where we use nested actions within the AS. An AS action makes a request leading to a *nested action* within the AS when the nested action is associated with a different transaction than the parent action. In the figure, action a_1 makes a request r_4 triggering action a_4 which is associated with a child transaction t_2 of t_1. The nested transaction can again be a global transaction spanning both AS and DB.

Thus, in the 1-N pattern, each client request/response pair is associated with a set of nested transactions. The transaction that is directly triggered by a client request and involves the action associated with the client request is called an *outer transaction*. For example, t_1 is the outer transaction directly triggered by request r_1. A nested transaction that is a child transaction of an outer

transaction is called an *inner transaction.* For example, t_2 is an inner transaction. An inner transaction is associated with a nested action while its parent transaction is associated with the corresponding parent action. A child transaction always terminates before its parent transaction; i.e., that is true nesting. An outer transaction can trigger a sequence of inner transactions. We can consider these inner transactions siblings as they have the same parent. Sibling transactions cannot be active concurrently, since each of them is triggered by a sub-request submitted by the action associated with the outer transaction and request execution is blocking. That is, one inner transaction has to complete before its sibling can start. An inner transaction itself can have child transactions which are also inner transactions. This leads to multiple levels of nesting where an inner transaction has not only a direct parent but can have a whole set of ancestor transactions. The outer transaction is ancestor of all inner transactions. An inner transaction is concurrent to all its ancestor transactions but while the inner transaction is executing, the ancestor transactions are suspended. A suspended transaction can only continue after all its descendants have terminated.

Please note, although many AS products allow the existence of nested transactions, most of them do not clearly define the rule for the relationship between parent transaction and nested transactions. J2EE regards a parent transaction and its nested transaction as two independent transactions, and the commit/abort of the parent transaction and the nested transaction will not affect each other. That is, the nested transaction can commit while the parent transaction aborts and vice versa. This is different to the traditional closed nesting model assumed in database systems. If this closed nesting model were applied, then all inner transactions and the outer transaction would commit at the same time at the very end. In order to somehow address this, we only consider relaxed state consistency. If a child transaction commits but the parent fails, it allows the parent transaction to adjust its state to reflect this fact.

Whether the parent transaction can see changes a child transaction performs on the shared state depends on the isolation level of the database system. The database system usually sees the different transactions as independent. For example, in case of serializability, the parent transaction can see changes of the child transaction after the child transaction commits. So can following sibling transactions. If any rule is required by an application, application developers should manage it at the application level.

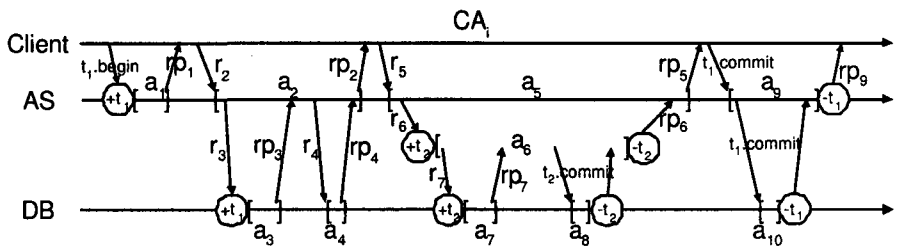


Figure 4.5: N-N pattern

Session state has usually no concurrency control applied. In principle, all transactions involved in a client request can access the same session state. As there are never two transactions executing at the same time, this appears fine. However, things become complex in case of abort. A transaction t_1 can change some session state and then trigger an inner transaction t_2 that reads these changes, performs further changes and then commits. If now t_1 aborts what is the semantics as t_2 has already read the "dirty" data and committed? As this is a very undesirable behavior, which should be avoided, we assume that nested transactions access disjoint session state. For instance, in J2EE, we only consider applications, where an inner transaction is executed on an SFSB that is not accessed by any of its ancestor transactions. Two sibling transactions, however, can access the same state as they execute serially.

4.2.4 N-N Pattern

The *N-N pattern* (N requests - N transactions) means that more than one client request can be executed within an outer transaction, and the execution can also trigger inner transactions. Typically, it is the mix of the N-1 pattern and the 1-N pattern. In an J2EE AS, if the user transaction and CMT is used together, and some EJB methods are configured as `RequiresNew`, then several client requests can be executed within one outer transaction, but sub-requests to methods with the `RequiresNew` attributes are executed within new inner transactions. Thus, the scenario follows the N-N pattern. Figure 4.5 illustrates such an N-N pattern. The client action CA_i first begins a transaction t_1, and then client requests r_2 and r_5 are executed as actions a_2 and a_5 within t_1. Additionally, a_5 triggers a transaction t_2. On the AS, t_2 is started by request r_6 that triggers action

a_6. The consecutive request r_7 leads t_2 to be started at the database. t_2 eventually commits at the end of a_6. a_6 is a nested action of a_5, and t_2 is a child transaction t_1. Finally, the client action CA_i submits the commit request $t_1.commit$ to commit transaction t_1 on both the AS and the database.

The N-N pattern also could have nested N-1 patterns. For example, when several client requests are executed within an outer transaction according to the N-1 pattern, the action associated with one of these requests starts an inner transaction and then submits several sub-requests within this inner transaction. It can happen in a J2EE AS when using the user transaction and BMT together. The user transaction provides the N-1 pattern for client requests. The BMT scheme allows an inner transaction to span more than one sub-request.

Due to the complexity of the N-N pattern, it is very difficult to correctly apply this pattern in a real application even without considering failure and replication. In practice, most of the time, the N-N pattern does not make sense. Applications who use it, probably are not aware of the implication of using such a complex pattern. If an application allows a client to explicitly bundle several requests into one transaction, it seems counter-intuitive that then one of these requests actively triggers several transactions. Hence, in this dissertation, we do not discuss the N-N pattern any further.

4.3 State Changes

We have seen all state changes at DB and AS are performed in the context of transactions. We now describe what it means that the state changes at AS and DB are consistent. If an AS/DB action changes the state of the AS/database, the action is called an *update action*. If a transaction involves one or more update actions, it is called an *update transaction*, otherwise a *read-only transaction*. Whether a DB action is an update action can be detected by analyzing the corresponding SQL statement. The state changed by an update DB transaction is the aggregation of state changes of all update DB actions involved in the DB transaction. At the AS, however, every AS action is assumed to be an update action, since updates are generally difficult to detect. This implies that every global transaction is assumed to be an update transaction in our model.

We consider two types of state at the AS: session-related state and shared state at the AS. As

we assume that shared state is cached data from the database, any change on the shared state at the AS is also reflected as changes of a DB action at the database. The change on the AS and the corresponding changes at the DB must belong to the same global transaction. Therefore, access to shared data at the AS is not considered part of an AS action but only considered part of a DB action. This means, we consider as state changed by an AS action only the changes performed on session-related objects. The state changed by an AS transaction is the aggregation of state changes of all AS actions involved in the transaction.

4.3.1 Transaction Histories to Reflect the Order of State Changes

In traditional serializability theory transactions are represented as a sequence of read and write operations. Serializability means that the interleaved execution of the operations of a set of transactions needs to be equivalent to a serial execution of the same set of transactions. Traditional concurrency control mechanisms such as strict 2-phase-locking and optimistic concurrency control furthermore have the property that if in the concurrent execution t_1 commits before t_2 then there exist an equivalent serial execution where t_1 also commits before t_2. Furthermore, no transaction ever reads uncommitted data (except of its own writes). Assuming that the DB provides such form of commit-order preserving serializability and given that the AS only changes session-related state where there are no concurrency issues, we can describe the order in which state changes occur at the AS and the DB through the order in which transactions commit.

At the AS (database), ATH (DTH) indicates the transaction history, i.e., the order in which AS (DB) update transactions commit. At initialization time, both the AS and the database are in the initial state, and both ATH and DTH are empty. After a transaction terminates, if it leaves state changes at the AS and/or the database, it is a *successful update transaction.* A DB update transaction is successful only when it commits, and an AS transaction is successful if it commits or, in case of relaxed state consistency, also if it aborts. A successful AS (DB) update transaction is appended to ATH (DTH) after it terminates. At any time, the visible state of the AS (database) is the aggregation of state changes made by all successful AS update transactions (successful DB update transactions) so far.

4.3.2 Matching State Changes at Application Server and Database

For a global transaction t, if state changes made by its DB transaction successfully commit, state changes made by its AS transaction must successfully commit as well. This relationship implies that each DB transaction in DTH must have a corresponding AS transaction in ATH. We assume that the DB and AS transactions of a global transaction have the same unique identifier, which allows them to be identified as being part of the same global transaction.

The ordering of transactions in ATH and DTH defines the ordering of state changes performed at the AS and the database. The ordering of transactions in DTH represents a serial order that is equivalent to the original concurrent execution of these transactions. Transactions can belong to different clients. The ordering of such transactions in ATH does not matter since changes made by them are performed on different session data and have no dependency. However, the ordering of transactions of the same client at the AS is important since it reflects possible dependencies. Furthermore, this order must be consistent with the order of the corresponding DB transactions. That means, given two global transactions t_1 and t_2 of the same client that both have AS and DB update transactions, if $AST(t_1) \prec AST(t_2)$ in ATH ($\prec$ representing the partial order in the history), then $DBT(t_1) \prec DBT(t_2)$ in DTH. There are two cases to consider. Given two transactions t_1 and t_2 of the same client, if t_1 and t_2 are not nested transactions, t_1 has to be completely executed either before t_2 or after t_2 since we assume there are no concurrent transactions triggered by the same client except for nested transactions. Otherwise, one transaction is the parent transaction of the other, and then the child transaction has to terminate before the parent transaction. We define the relationship between ATH and DTH in form of a matching property:

Definition 4.3.1. *We say ATH and DTH match (denoted as $ATH \bowtie DTH$) if the following holds:*

1. $\forall t \in DTH$: *eventually* $AST(GTX(t)) \in ATH$.
2. $\forall t \in ATH$ *and*
   - (a) *t commits: if* $DBT(GTX(t))$ *is an update transaction,* $DBT(GTX(t)) \in DTH$.
   - (b) *t aborts (possible in case of relaxed state consistency):* $DBT(GTX(t)) \notin DTH$.

3. *Given* $t_1, t_2 \in DTH$ *and* $CL(GTX(t_1)) = CL(GTX(t_2))$: $DBT(t_1) \prec DBT(t_2)$ *in* $DTH \Leftrightarrow AST(GTX(t_1)) \prec AST(GTX(t_2))$ *in* ATH.

Definition 4.3.1 expresses the requirement that transactions and request executions at the AS and the DB must match and are executed in a consistent order at both tiers.

4.3.3 Matching State Changes at Application Server and Client Request Execution

Definition 4.1.1 has indicated how requests must be properly associated with actions and corresponding responses. Definition 4.3.1 relates the state at the AS and DB tier via transactions. As a final property, we relate the requests at the client tier with the proper state at the AS and thus, indirectly with the proper state at the DB. The state changes performed by AS transactions should be consistent with client requests associated with these transactions. In our model, this consistency is expressed by a matching between ATH and RH_{CA_i}/RPH_{CA_i} of client session CA_i and is denoted as $ATH \bowtie RH_{CA_i}/RPH_{CA_i}$. Matching not only means the content of ATH and RH_{CA_i}/RPH_{CA_i} match, namely each request/response in RH_{CA_i}/RPH_{CA_i} has at least one associated transaction, but also means the ordering of ATH and RH_{CA_i}/RPH_{CA_i} match, namely transactions must be ordered according to request execution.

Since the association between client requests and transactions are different for different execution patterns, the definition of the matching rules depends on the execution pattern. Let's have a look at each of them individually,

1-1 Pattern

Each client request r_i and the triggered action $a \in A(r_i)$ is associated with exactly one transaction. This means that, on the one hand, for each successful update transaction $t \in ATH$, the only action a involved in t should have its corresponding request $R(a) \in RH_{CA_i}$ of a client C_i and have its corresponding response $RP(a) \in RPH_{CA_i}$ of the same client session CA_i. On the other hand, for each request $r \in RH_{CA_i}$ of a client session CA_i, ATH must eventually contain exactly one transaction t that is associated with an action $a \in A(r)$, unless the request has an abort response in RPH_{CA_i} in case of full state consistency. In case of full state consistency and an abort response,

ATH must not contain the transaction. For any two requests in RH, their ordering in RH should be the same as the ordering of the corresponding AS transactions in ATH. We can formalize this as follows.

Given AS transaction t, $at(t)$ indicates the AS action involved in t. The matching property between ATH and RH_{CA_i}/RPH_{CA_i} of client C_i is called *1-1 matching property*.

Definition 4.3.2. *$ATH \bowtie RH_{CA_i}/RPH_{CA_i}$ if the following holds:*

1. $\forall t \in ATH \wedge CL(GTX(t)) = C_i$: *eventually* $R(at(t)) \in RH_{CA_i} \wedge RP(at(t)) \in RPH_{CA_i}$.
   (a) *In case of full state consistency,* $RP(at(t)) \neq rp_{ab_t}$.
   (b) *In case of relaxed state consistency, t aborts* $\Leftrightarrow RP(at(t)) = rp_{ab_t}$.
2. $\forall r \in RH_{CA_i}$:
   (a) *In case of full state consistency, either eventually* $\exists t, t \in ATH \wedge r = R(at(t))$, *or eventually* $\exists rp \in RPH_{CA_i}, POS^{r}_{RH_{CA_i}} = POS^{rp}_{RPH_{CA_i}} \wedge rp = rp_{ab_t}$.
   (b) *In case of relaxed state consistency, eventually* $\exists t, t \in ATH \wedge r = R(at(t))$.
3. *Given* $t_1, t_2 \in ATH \wedge CL(GTX(t_1)) = CL(GTX(t_2))$: $t_1 \prec t_2$ *in* $ATH \Leftrightarrow R(at(t_1)) \prec R(at(t_2))$ *in* RH_{CA_i}.

Condition 1 captures that each successful update transaction has a matching request and response. Condition 2 captures that each client request of client C_i has a matching successful update transaction on the AS unless it is aborted in case of full state consistency, in which case the client receives an abort response. Condition 3 captures that transactions must be ordered in ATH according to the ordering of their corresponding requests. It actually also guarantees that neither two transactions are associated with the same request nor two requests are associated with the same transaction.

N-1 Pattern

In this pattern, serval client requests and their actions are associated with a transaction. On the one hand, for each transaction $t \in ATH$, each action a involved in t should have its corresponding

request in RH_{CA_i} of a client session CA_i and have its corresponding responses in RPH_{CA_i} of the same client session. Furthermore, the requests (responses) of actions associated with a transaction t should build a consecutive sequence in one RH_{CA_i} (RPH_{CA_i}). Additionally, for two transactions t_1 and t_2 of the same client session CA_i, if $t_1 \prec t_2$ in ATH, then all requests (responses) of actions associated with t_1 are before all requests (responses) of actions associated with t_2 in RH_{CA_i} (RPH_{CA_i}). On the other hand, in RH_{CA_i} of a client session CA_i, if a sequence of requests belongs to the same transaction, the transaction must be eventually contained in ATH exactly once, unless the last request in this sequence has a corresponding abort response in RPH_{CA_i} in case of full state consistency, in which case ATH should have no transaction associated with these requests.

To formalize this, we need some further notation. Given an AS transaction t, the function $AT(t)$ represents the sequence of AS actions that are associated with transaction t. Due to the blocking scheme, the actions are in a sequential order. For example, if t contains actions a_1 to a_n, then $AT(t) = a_1 a_2 ... a_n$. $AT^i(t)$ represents the ith action in the sequence.

We now apply functions R and RP to an action sequence to represent the request sequence that triggers the action sequence and the response sequence generated by the action sequence. Thus, $R(AT(t)) = R(a_1)R(a_2)...R(a_n)$ represents the sequence of client requests associated with transaction t and $RP(AT(t)) = RP(a_1)RP(a_2)..RP(a_n)$ represents the sequence of responses generated within transaction t. These two sequences must respectively be sub-sequences of the request history and the response history of the client $CL(GTX(t))$ that triggers the transaction for the matching between ATH and RH/RPH of the client.

In order to formally express the relationship of sequences, we define the following notations for sequences.

- $|s|$ indicates the size of sequence s.
- s^k $(1 \leq k \leq |s|)$ indicates the kth item of sequence s.
- $s' \propto s$ indicates that sequence s' is a sub-sequence of sequence s.
- $s_1 \prec s_2$ indicates that the last item of sequence s_1 precedes the first item of sequence s_2.

With this, the fundamental property of ATH and RH/RPH to match is that for a given AS transaction t, $R(AT(t)) \propto RH_{CL(t)}$ and $RP(AT(t) \propto RPH_{CL(t)}$.

Formally for the N-1 pattern, the matching property between ATH and RH_{CA_i}/RPH_{CA_i} of client C_i is called *N-1 matching property*, and is defined as follows.

Definition 4.3.3. *$ATH \bowtie RH_{CA_i}/RPH_{CA_i}$ if the following holds:*

1. $\forall t \in ATH \wedge CL(GTX(t)) = C_i$: *eventually* $R(AT(t)) \propto RH_{CA_i} \wedge RP(AT(t)) \propto RPH_{CA_i}$.
   (a) *In case of full state consistency,* $RP(AT^k(t)) \neq rp_{ab_t}$ $(1 \leq k \leq |AT(t)|)$.
   (b) *In case of relaxed state consistency,* t *aborts* $\Leftrightarrow RP(AT^k(t)) \neq rp_{ab_t}$ *for* $1 \leq k < |AT(t)|$, *and* $RP(AT^k(t)) = rp_{ab_t}$ *for* $k = |AT(t)|$.
2. $\forall r \in RH_{CA_i}$:
   (a) *In case of full state consistency, either eventually* $\exists t, t \in ATH \wedge r \in R(AT(t))$, *or eventually* $\exists rs \propto RH_{CA_i}, r \in rs \wedge POS^{rs|rs|}_{RH_{CA_i}} = POS^{rp}_{RPH_{CA_i}} \wedge rp = rp_{ab_t}$.
   (b) *In case of relaxed state consistency, eventually* $\exists t, t \in ATH \wedge r \in R(AT(t))$.
3. *Given* $t_1, t_2 \in ATH \wedge CL(GTX(t_1)) = CL(GTX(t_2))$: $t_1 \prec t_2$ *in* $ATH \Leftrightarrow R(AT(t_1)) \prec R(AT(t_2))$ *in* RH_{CA_i}.

Condition 1 captures that each successful update transaction has a sequence of matching requests and responses. Condition 2 captures that a client request has a matching successful update transaction on the AS unless the transaction is aborted in case of full state consistency, in which case the last client request associated with the transaction has an abort response. Condition 3 captures that transactions must be ordered in ATH according to the ordering of request sequences associated with these transactions. In fact, it also guarantees that two transactions are not associated with the same request.

1-N Pattern

In this pattern, the execution of a client request might be associated with more than one transaction. The relationship between a client request, its action and the corresponding outer transaction is the same as the 1-1 pattern. For each inner transaction in ATH, there is exactly one outer transaction in ATH that is the ancestor of the inner transaction. We assume there is always an outer transaction as we only consider relaxed state consistency (see Section 4.2.3). The ordering of two requests in RH_{CA_i} of a client session CA_i should be the same as the ordering of the corresponding outer transactions in ATH if applicable, just as in the 1-1 pattern. An inner transaction must always precede the parent transaction in ATH since the child transaction always terminates before its parent transaction. Transitively, any inner transaction must precede its outer transaction in ATH.

As it is more complex to define these properties formally, we make a formal description only in Section 6.2.

4.4 Correct Request Execution

In conclusion, the standard behavior of a non-replicated non-faulty AS can be described by three matching properties.

Definition 4.4.1. *Given an execution in a 3-tier system with client sessions CA_i, $1 \leq i \leq n$. Let ATH be the transaction history at the AS, and DTH be the transaction history at the database. The standard behavior of a non-replicated non-faulty AS has the following three properties:*

1. $\forall i, 1 \leq i \leq n$: $RH_{CA_i} \bowtie RPH_{CA_i}$,
2. $\forall i, 1 \leq i \leq n$: $ATH \bowtie RH_{CA_i}/RPH_{CA_i}$,
3. $ATH \bowtie DTH$.

The first property captures the exactly once execution of client requests as perceived by clients. The second property captures the exactly once execution of client requests as it really happens on the AS. The proper association between requests, responses and transactions is made. The third

property captures the consistency of state changes on the AS and the database at a per transaction basis.

Compared with the traditional correctness criteria introduced in Section 3.2, our correctness criteria consider the global picture of a multi-tier architecture, focusing on the relationship between different tiers and how execution proceeds across these tiers. This is motivated by the fact that a crash of one tier not only affects the crashed tier itself but also affects other tiers linking to the crashed tier. For each tier,we define the behavior that makes sense:

1. Clients receive proper responses.
2. State changes at the AS and their relative order reflect the order in which requests are submitted and how requests are associated with transactions.
3. State changes at the database and the AS are consistent.

What we ignore so far is the issue of serializability or potentially other isolation levels. We assume the database to provide serializability and the AS not to require concurrency control as access to shared data is synchronized via the central database.

4.5 How a Crash Affects Correctness

A failure on the AS has two implications. Firstly, ATH becomes Λ, namely empty, indicating that the crashed AS loses its state. As a result ATH and DTH do not match anymore. While ATH is now empty, DTH still contains all committed update DB transactions, since persistent data is not affected by the crash and only all ongoing DB transactions are automatically aborted at the database. Furthermore, ATH does not match any more with RH/RPH of ongoing client sessions, since these RH/RPH remain as before the crash while ATH is empty. The second implication of a failure is that all outstanding client requests do not receive their expected responses. This violates the matching requirement of RH_{CA_i} and RPH_{CA_i} of a client session CA_i.

4.6 Correctness of Passive Replication

In passive replication, a primary replica accepts and executes the client requests and propagates state changes to the backup replicas where they are applied. Depending on the replication strategy, state propagation can take place at different time points. When the primary crashes, a backup takes over as new primary and continues execution. The correctness criteria summarized in Definition 4.4.1 can be easily extended to reflect a replicated system using passive replication by requiring any change in primary not to be visible at the client, and the current primary replica to fulfill the consistency requirements.

Definition 4.6.1. *Given an execution in a 3-tier system with client sessions CA_i, $1 \leq i \leq n$. Let AS $= \{AS_j, 1 \leq j \leq m\}$ be the set of AS replicas in the middle-tier, ATH_i be the transaction history at AS replica AS_i, and DTH be the transaction history at the database. The execution is correct if*

1. $\forall i, 1 \leq i \leq n$: $RH_{CA_i} \bowtie RPH_{CA_i}$
2. *Let $AS_j, 1 \leq j \leq m$ be the current primary:* $\forall i, 1 \leq i \leq n$, $ATH_j \bowtie RH_{CA_i}/RPH_{CA_i}$
3. *Let $AS_j, 1 \leq j \leq m$ be the current primary:* $DTH \bowtie ATH_j$.

The challenge of providing a correct replication algorithm lies in the fact that the different tiers and the replicas communicate via asynchronous messages and histories get updated at different time points. For instance, in the 1-1 pattern, a client receives the response after the transaction AS transaction terminates. The DB transaction always terminates before its corresponding AS transaction. The primary replicates the changes performed by a transaction either before the transaction commits or after the transaction commits. A crash can occur at any time, and thus, this asynchrony between events can leave the system in an inconsistent state that has to be resolved before execution can continue at the new primary.

Chapter 5

ADAPT-SIB Replication Algorithm for 1-1 Pattern

In the next two chapters we present the replication tool ADAPT-SIB which implements replication algorithms for the various execution patterns. ADAPT-SIB focuses on fault-tolerance, but, as it uses passive replication, and thus avoids redundant computation, it has the potential to be integrated into an architecture where replicas are also used for scalability and load-balancing (which will be the topic of Chapter 8).

ADAPT-SIB uses eager and warm replication for session-related state, propagating state changes performed by a transaction from the primary to the backup replicas before the transaction terminates. Shared state is synchronized via the database. In order to provide a generic and practical solution, ADAPT-SIB does not require any special support from clients or the database.

As mentioned in the last chapter, the behavior of the AS can be categorized by different execution patterns, which associate client requests and transactions in different ways. This chapter presents a replication algorithm for the 1-1 pattern and proves its correctness.

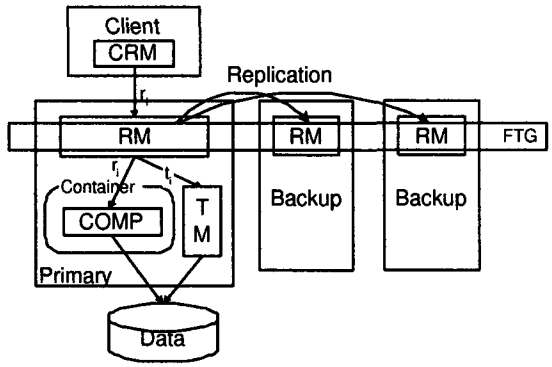


Figure 5.1: Architecture of ADAPT-SIB

5.1 Structure of ADAPT-SIB

Figure 5.1 shows the architecture of ADAPT-SIB. ADAPT-SIB assumes that a cluster of AS replicas consists of one primary replica and several backup replicas. Each AS replica has a *replication manager* (RM). The RM uses the group membership primitive of the GCS to maintain a fault tolerance group (called FTG). The RM also uses the multicast primitive of the GCS to send messages within the FTG. The replication algorithm has *client, primary, backup* and *failover* parts. The primary algorithm is executed at the RM of the current primary AS. We assume the replication tool obtains control before a request is sent to the TM (transaction manager) or a component, and after the call returns. The backup algorithm is executed at the RM of each backup replica, and the failover algorithm is executed at the RM of the backup that is selected as the new primary after the crash of an old primary. At the client, there is a *client replication manager* (CRM) that runs the client part of the replication algorithm. It intercepts each client request and response at the client side. For web clients, CRM actually resides in the web server. As mentioned in Section 2.3.2, in J2EE the client needs to create a connection session with the AS to get the stub of a targeting EJB object. In our implementation solution, the CRM object is created when the connection session is built, and is downloaded to the client side together with the stub of the EJB object from the server side.

Here is the basic idea of the algorithm. Assuming the fault tolerance group contains m replicas. A client request r is executed at the current primary. Changes on session-related data performed

within a transaction t is recorded. If the current primary does not crash, at the commit time of the transaction t, recorded state changes and the response to the client are propagated to backups, and only then the transaction is committed. No changes on shared data are sent since they are written to the shared database. Backups only apply the state changes after they know that transaction t has actually committed. If the current primary crashes before the client receives r's response rp_r, the CRM sends outstanding r to the new primary which is chosen from the remaining available backups. The actions of the new primary at the time of failover depend on the state changes it has received, and the set of transactions that successfully committed at the database.

Recall the correctness criteria proposed in Theorem 4.6.1. Let's analyze the most common situations after the old primary crashes, the AS replica AS_j is selected as the new primary and the new primary AS_j receives the resubmitted request r. If AS_j has already received the state changes of the transaction associated with the request r and the database transaction also committed, it does not reexecute the request. Instead it applies the state changes and returns the response. With this, the request has one matching response (Definition 4.1.1 (1)) and one matching transaction at the current primary AS (Definition 4.3.2 (2)), which has one matching transaction at the DB (Definition 4.3.1 (2)). If AS_j has received the state changes but the database transaction did not terminate properly before the crash (note that this can be possible because the primary sends the changes eagerly), then it may not apply the AS state changes. If it did, ATH_j would no more match DTH (due to 4.3.1 (2a)). Instead, it discards the state changes and starts request execution from scratch to have exactly-once execution across all tiers. If it hasn't received the state changes, it knows that the database transaction has not committed. Thus, neither ATH_j nor DTH contain a transaction associated with the request. Thus, it also starts request execution from scratch. This behavior guarantees that neither the AS transaction nor the DB transaction is executed twice or that one or both of the transactions are missing.

The main data structures used in the pseudo code of all algorithms of ADAPT-SIB are as follows. *Request*, *Response*, and *Component* are encapsulated in corresponding objects. A transaction is identified by a unique identifier $txid$ of type TID. The server maintains an EU object for each currently active transaction (one per client). EU keeps track of transaction identifier $txid$, the set of components $COMP$ that have been accessed so far, the pair of the client request req and its

```
Response invoke (Request req, Component comp)
  1. Generate req.rid;
  2. while (true)
  3.   Response resp = primary.invoke(req, comp, nil);
  4.   if (∄ failure Exception) return resp;
  5.   else find a new primary;
(a) client replication algorithm

TID begin ()
  1. new EU eu;
  2. eu.txid = TM.begin();
  3. return eu.txid;
(b) primary: intercept begin transaction request to TM

Response invoke (Request req, Component comp, TID txid)
  1. if (∃ (req.rid, resp) ∈ RR)
  2.   TM.abort(txid);
  3.   return resp ;
  4. if (req is a client request) eu.req = req;
  5. eu.COMP ∪ = {comp};
  6. Response resp = comp.invoke(req);
  7. if (req is a client request)
  8.   eu.resp = resp;
  9.   RR ∪ = {(req.rid, eu.resp)};
 10.   if (resp == abort exception) abort_proc();
 11. return resp;
(c) primary: intercept request to component

void abort_proc ()
  1. eu.COMP = ∅;
  2. new aborted Message m3;
  3. m3.content = {eu};
  4. multicast m3 by reliable delivery;
(d) primary: abort procedure

void commit (TID txid)
  1. for each comp ∈ eu.COMP
  2.   set comp.state to current state of
       corresponding component;
  3. new committing Message m1;
  4. m1.content = {eu};
  5. if ( the current transaction updated the database)
  6.   eu.db = true;
  7. multicast m1 by uniform reliable delivery;
  8. if (eu.db == true) insert eu.txid into database;
  9. wait until receive m1;
 10. TM.commit(txid);
 11. if (∄ abort Exception and eu.db == true)
 12.   new committed Message m2;
 13.   m2.content = {eu.txid};
 14.   multicast m2 by reliable delivery;
 15. else if (∃ abort Exception)
 16.   eu.resp = abort exception;
 17.   abort_proc();
(e) primary: intercept commit transaction
request to TM
```

Figure 5.2: 1-1 Algorithm at the client and primary

response *resp*, and the flag *db* to mark whether or not the transaction updates the database. The *Message* object represents messages between replicas. The *content* of a *Message* object depends on the type of message. Further data structures will be introduced later.

5.2 1-1 Replication Algorithm for Full State Consistency

The algorithm for the 1-1 pattern supporting full state consistency is from [109]. Some of the ideas are based on [43, 41].

Figure 5.2 (a) shows that the client replication algorithm intercepts each request submitted from the client to the server. It attaches a unique id, and forwards the request to the current primary (lines 1 and 3). Upon a failure exception, it resends the request with the same id to the new primary. This repeats until it receives a correct response (lines 2, 4, and 5).

At the server site, each replica maintains a set RR of past request/response pairs which is needed to avoid duplicate request execution. The execution associated with a client request happens within a single execution thread. We assume the replication algorithm intercepts a client request and any further requests made to components. Furthermore, it intercepts begin and commit requests made to the TM. For simplicity of description, we assume that transactions are server managed (CMT in J2EE terminology), i.e., the container starts a transaction upon a client request before any component method is called, and commits the transaction after all component execution has finished. Other types of transaction management are conceptually the same, but would require a different notation in description. Using container managed transactions, the begin transaction command submitted to the TM is the first call intercepted by the replication algorithm for a client request r, and the commit call is the last one. Upon intercepting a begin transaction request (Figure 5.2 (b)), an eu object is created and associated with the thread before the begin is forwarded to the TM. Upon intercepting a request to a component (Figure 5.2 (c)), the algorithm first checks whether the request was already successfully executed. This can happen when the old primary executed the request successfully, informed the backups and committed the transaction but crashed before returning the response. In this case, the client algorithm resubmits the request to the new primary. The new primary, however, has the response for this request stored in RR, and no new execution is triggered. The transaction that has been associated with the thread is aborted, and the response immediately returned (lines 1-3). Otherwise, if the request is a new client request, it is recorded in eu (line 4). Furthermore the component to be accessed is recorded in eu before the request is forwarded to the corresponding component (lines 5-6). Recall that there can be nested calls to different components, all within the same transactional context. Each of them is intercepted, and the component information added to the corresponding eu. If an abort takes place during execution of any request (client request or nested sub-requests), an abort exception will be returned as response. In the case of an abort within a nested request, this abort exception is simply forwarded upwards along the calling hierarchy until it reaches the client request. Note also that each component rolls back its state changes associated with the transaction (full state consistency). When the execution of the client request completes, the server first records the response in eu and records the pair of request and response (it is an abort exception in case of abort) in RR (lines 8-9). In case an abort occurred during execution, an abort

procedure (Figure 5.2 (d)) is called (line 10) which informs the backups about the abort. Since full state consistency is assumed, no state changes need to be transferred within this aborted message. Finally, the response is returned (line 11). Upon intercepting the commit transaction request (Figure 5.2 (e)), a *committing* message is multicast using uniform reliable delivery. The message includes the final state for each accessed component and the pair of the client request and its response (line 1-4). While waiting for its uniform reliable delivery, the *txid* is inserted into the database if this is a DB update transaction (lines 8). This will help backups to determine whether a transaction has actually committed at the database or not if in-doubt. After the primary receives its own committing message, it commits the database transaction (line 10). As uniform reliable delivery is used, receiving the own committing message is equivalent to receiving from all backups a confirmation that they have received the message. Once commit was successful, the primary multicasts a *committed* message (lines 12-14) if the transaction updated the database, and the commit procedure completes. The committed message makes the backups be aware of the commit of the transaction. Note that, in theory, the database might abort the transaction upon receiving the commit request. However, when only one database system is accessed, this usually does not happen (it might happen if the database uses optimistic concurrency control, but this is not the case for current relational databases). This special abort case can be handled sending an abort message as if abort occurs during normal processing (line 16-17).

All messages the primary sends to the backups use FIFO ordering. The backups, during normal processing, store all received messages in a FIFO queue. Furthermore, if clients connect to them while they are not primary (the GCS has not delivered a view change message and thus they have not determined that they are the new primary), they respond to the client that they are not the primary. If the GCS delivers a view change message indicating that the primary was excluded from the group, one of the backups is selected as the new primary. This could be decided by a pre-defined priority list or by an election procedure [59]. Any message from the now crashed primary, that the GCS delivers after the view change, is ignored by all surviving replicas. The new primary now starts failover (Figure 5.3). Committing messages are processed in FIFO order to track the latest state of each component (lines 2-3). The procedure first checks whether the corresponding database transaction committed or aborted (lines 4-5) if it is a DB update transaction. The DB transaction committed for

```
void failover ()
 1. new Eu eu, new set COMP;
 2. in order of reception process each committing message m
 3.   eu = m.content;
 4.   if (eu.db == true and ∄ committed message m' with m'.content.txid == eu.txid and eu.txid
      does not exist in database)
 5.         ignore committing message
 6.   else // transaction committed
 7.     for each comp ∈ eu.COMP
 8.       if (∃ c ∈ COMP and c == comp)
 9.         c.state = comp.state;
10.       else COMP = COMP ∪ {comp};
11.     RR = RR ∪ {(eu.req.rid, eu.resp)};
12. for each aborted message m
13.   eu = m.content
14.   RR = RR ∪ {(eu.req.rid, eu.resp)};
15. for each comp ∈ COMP
16.   create corresponding component;
17.   set component's state to comp.state;
```

Figure 5.3: 1-1 failover

sure if the new primary received the committed message but also if the $txid$ marker can be found in the database, as only committed markers remain in the database. In the case of abort, the committing message is ignored (line 6). Otherwise, the procedure determines the affected components, and records the pair of the client request associated with the transaction and its response in RR (lines 8-12). This is then used to detect duplicate requests. For each aborted message, we record the pair of the client request and the corresponding abort response in RR (lines 14-15). Finally, all necessary components are recreated (16-18). In Section 7.1, we discuss alternative failover strategies in more detail. Note that for clarity, the algorithm does not contain obvious garbage collection actions, such as keeping for each client only the last request/response pair as it will, if at all, only resubmit the last outstanding request.

5.3 Correctness

This section formally proves the correctness of the proposed algorithm for full state consistency by showing that the algorithm fulfills all matching properties described in Theorem 4.2.1.

5.3.1 Successfully Completed Requests

Assume that at the start of the system, AS_o is primary. As long as there is no crash it is obvious that all properties are fulfilled since they have been defined to model a non-faulty environment. Given client session CA_i, suppose the client has submitted so far $x-1$ requests and received $x-1$ responses. Thus, $RH_{CA_i} = r_1 r_2 \ldots r_{x-1}$, $RPH_{CA_i} = rp_1 rp_2 \ldots rp_{x-1}$. As the client is blocking, if it has received rp_{x-1} it is guaranteed to have received all $rp_1 \ldots rp_{x-2}$. Each request r_h started a transaction t_h but as some might abort, the projection of ATH_o on transactions of client session CA_i is $ATH_o^i = (AST(t_1) \vee \perp)(AST(t_2) \vee \perp) \ldots (AST(t_{x-1}) \vee \perp)$ where $\perp$ refers to the ATH having no entry for that specific transaction as it aborted. Similarly, the projection of DTH on transactions of client session CA_i is $DTH^i = (DBT(t_1) \vee \perp)(DBT(t_2) \vee \perp) \ldots (DBT(t_{x-1}) \vee \perp)$. For each $AST(t_k) \notin ATH_o$, $DBT(t_k) \notin DTH$ but there might be $DBT(t_k) \notin DTH$ where $AST(t_k) \in ATH_o$ as some committed transactions might not have a database transaction or the database transaction is read-only and is not registered in DTH. As we assume the client to be blocking and the database to provide prefix-committed serializability, the order of requests in RH_{CA_i}, their corresponding responses in RPH_{CA_i}, and their corresponding committed transactions in ATH_o and DTH is the same.

The backup AS_j receives committing messages in FIFO order and puts them in a queue MQ_j. Projected on client C_i the queue eventually contains for each committed transaction t the message $committing_t$. The uniform reliable delivery and the FIFO order guarantee that both the content and the ordering of transactions in MQ_j match those of ATH_o (denoted as $MQ_j \bowtie ATH_o$). Note that according to the discussion above it is guaranteed that when the client receives a response for a committed transaction, MQ_j contains the corresponding committing message.

Now assume AS_o crashes after rp_{x-1} was returned. After the crash the new primary AS_j installs state changes of committing messages in MQ_j according to the ordering. For each of the committing messages (i) there is either no DB update transaction and the changes are installed, (ii) a committed message was received and the changes are installed, (iii) an aborted message was received and the changes discarded, or no committed/aborted message is received and AS_j checks in the database for the transaction identifier. If it determines the database transaction committed it

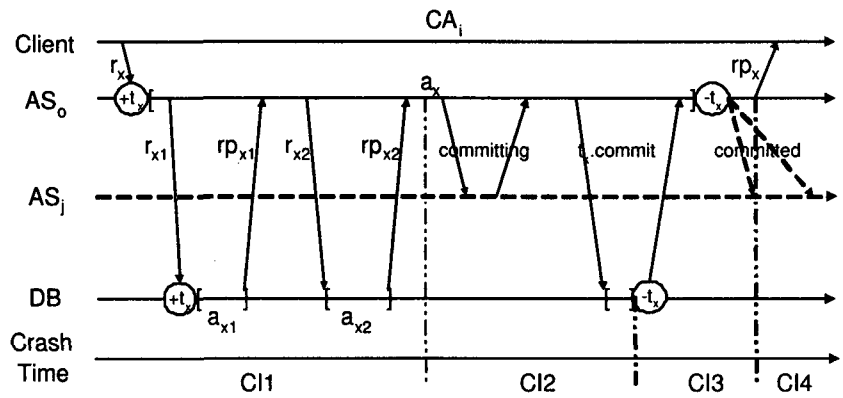


Figure 5.4: Possible crash intervals of the 1-1 algorithm in case of a commit

applies the changes, otherwise it discards them. In any of the three cases above, changes are exactly then installed and with it the AS transaction appended to ATH_j if the corresponding AS transaction at the old primary was appended to ATH_o before the crash. That is, in regard to client session CA_i, ATH_j has now the same sequence of transactions as ATH_o had before the crash. Thus, correctness up to request r_{x-1} is given.

5.3.2 Crash during Request Execution

Now we look when AS_o crashes just around the time at which the client session CA_i submits request r_x. Assume AS_j becomes the new primary. Our approach to analyze the behavior is to enumerate different "crash intervals", i.e., time intervals in which a crash requires different actions from the system in order to guarantee correctness. As the order of certain events might be different from execution to execution, our sequence of crash intervals is only an approximation to help structure the proof.

Commit case Figure 5.4 extends the execution in case of a commit for the 1-1 pattern shown in Figure 4.2. It shows client C_i, the primary AS_o, the backup AS_j, and the DB. Furthermore, it has an axis depicting different crash intervals at which the failover needs to perform different actions. In the figure, the client submits request r_x. When receiving the request, the primary AS_o

starts action a_x and at the same time global transaction t_x. It sends two sub-requests, r_{x1} and r_{x2} to the database, which are executed within the same transaction t_x. Just before the commit, it multicasts the committing message with state updates and the response to the backups. Only when it has received its own message, the database transaction, if it exists, commits (and after having written the transaction identifier into the database if it was an update transaction). Then, the AS primary commits the AS transaction and sends the commit message (if there was an update database transaction), before returning the answer to the client.

For depicting the crash intervals, we assume that when a message is sent, it will also be received. If the primary sends a message, but the message is lost and the primary crashes before the GCS at the primary resends it, we consider this a crash before message send. The first crash interval (CI1) ends just before the primary sends the committing message as the backups have no knowledge about the transaction before receiving the committing message. In the figure we have modeled the uniform reliable delivery as a message/acknowledgement pair between primary and backup. In principle, using uniform reliable delivery, the primary can receive its own committing message before or after it is received at the backups. However, the GCS guarantees that if the primary receives the message, then the backups will not only also receive the message but, in case of a crash of the primary, they will also receive it before they receive the view change message excluding the primary from the group. Furthermore, either they all receive it before the view change, or none receives it, or they all receive it after the view change. The latter messages are ignored by all. Thus, from a logical perspective, this uniform reliable delivery is equivalent to an acknowledge-based propagation: the primary sends the committing message, the backups send confirmations and only when the primary has received all these confirmations, it continues with the commit. The figure depicts this logical ordering of messages. The second crash interval (CI2) starts after the primary has sent the committing message (potentially allow the backups to add the transaction to their ATH) and ends just before the commit of the database transaction (which adds this transaction to DTH). The third crash interval (CI3) starts with the database commit (which adds this transaction to DTH) and ends just before the primary sends the response to the client. The fourth interval (CI4) starts after the response is sent (which allows the client to add it to RPH).

Let's now discuss what happens if the primary AS_o crashes while being in one of these crash

intervals.

CI1: The CRM receives a failure exception. AS_j has not received the *committing* message and has no knowledge about the request or the associated transaction. The database transaction, if it exists, is aborted upon the crash of AS_o. Thus, $AST(t_x) \notin ATH_j$ and $DBT(t_x) \notin DTH$. The CRM resubmits the request to AS_j where it is executed as a completely new request leading to exactly one execution within the transaction t'_x, eventually returning rp_x as the last response so far in RPH_{CA_i}, and possibly adding $AST(t'_x)$ to ATH_j and $DBT(t'_x)$ to DTH as the last transactions for client session CA_i so far. All matching requirements are fulfilled.

CI2: The CRM receives a failure exception. The database transaction, if it exists, aborts upon the crash. Thus $DBT(t_x) \notin DTH$. AS_j has already received the *committing* message for t_x. There are two cases to consider. First, if $eu.db = false$, then AS_j adds $AST(t_x)$ to ATH_j at failover. When CRM resubmits the request, AS_j immediately returns the response. Correctness is given, as request, response and ATH match. DTH does not contain a transaction but this is fine, as the original execution at AS_o was completed and did not involve database updates. In the second case, i.e., $eu.db = true$, AS_j checks the database for the transaction identifier at failover. It can't find it since the database transaction did not commit. However, AS_j knows that an update DB transaction was involved. If it appended $AST(t_x)$ to ATH_j, a mismatch would occur (according to Definition 4.3.1 (2a)). Thus, AS_j discards the content of the *committing* message. Therefore, neither ATH_j nor DTH have transactions related to r_x. When the CRM resubmits the request, AS_j executes it as a completely new request leading to exactly one execution and one response as discussed in $CI1$.

CI3: The CRM receives a failure exception and $DBT(t_x) \in DTH$. At AS_j, the *committing* message was received. At failover, AS_j detects that $DBT(t_x)$ has committed because it either has already received the *committed* message or it has looked for and found the transaction identifier in the database. Thus, $AST(t_x)$ is added to ATH. Thus, ATH_j already matches the request history and DTH. When the CRM resubmits r_x, AS_j immediately returns the response, completing the matching requirements.

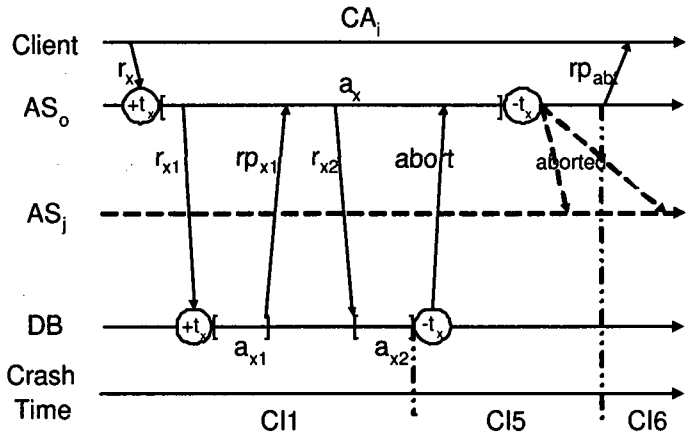


Figure 5.5: Possible crash intervals of the 1-1 algorithm in case of an abort during execution

CI4: From the perspective of AS_j this is the same as $CI3$ and $AST(t_x)$ is added to ATH. The only difference is that the client receives the response from the old primary. Thus, there is no resubmission.

Aborts Now let's have a look at the abort case. There are three cases to consider. First, the AS initiates the abort for some reason. Second, the database triggers an abort some time during execution. Third, the database aborts when the AS submits the commit request. The first two are handled in the same way in our algorithm as the AS only sends the abort message after the database aborts. Figure 5.5 shows the situation where a_{x2} leads to the abort at the database. In this case, the first crash interval ends just before the database transaction aborts. As the situation is exactly as for crash interval $CI1$ of the commit case, we do not discuss it further. Then, the AS receives an abort response from the database, and the AS transaction also aborts. After the state is rolled back, an *aborted* message is sent to the backups and then the abort response is returned to the client. As the abort message is only sent with reliable delivery, the reception of the message at the backups (if at all) and the response to the client can be in any order. If there is a crash, it could be that both the abort response to the client and the abort message to the backups were received, none was received, or only one of them was received. In regard to crash intervals, we consider crash interval $CI5$ to end just before the old primary sends the response to the client, and $CI6$ after sending this message.

Let's now consider the actions upon a crash during these crash intervals:

CI5: The CRM receives a failure execution. As the DB transaction aborts, $DBT(t_x) \notin DTH$. For new primary AS_j we can consider two cases.

(a) AS_j has not received the *aborted* message before the crash. When CRM resubmits r_x, AS_j will reexecute as if it were a new request as AS_j does not know anything about t_x. Although there are now two executions this is correct as the first execution did not leave any entries in either ATH_j or DTH and no response to the client. The second execution might again lead to an abort, with no transaction in ATH_j or DTH and an abort response, or execution might succeed with a commit and the corresponding response. At this time, all histories do match.

(b) AS_j has received the *aborted* message. In this case, when CRM resubmits r_x, the abort response rp_x is immediately returned. Request, response, ATH and DTH histories match with both AS and DB transactions aborted.

CI6: $DBT(t_x) \notin DTH$ as the DB transaction aborts. $AST(t_x) \notin ATH_j$, either because AS_j has received the *aborted* message or because it hasn't received any message at all. In the first case AS_j adds the request/response pair to RR, otherwise not. But this difference has no effect, as the CRM will not resubmit the request since it already received the abort response. All histories match.

Finally, Figure 5.6 shows what happens if the database transaction aborts upon the commit request submitted by the AS. As in the commit case, crash interval $CI1$ ends just before the primary AS_o sends the *committing* message, and $CI2$ ends just before the database transaction terminates. $CI7$ now ends just before the primary AS_o sends the client response, and $CI8$ starts after sending this message.

CI7: The CRM receives a failure exception. The database transaction has aborted because of application semantics, thus, $DBT(t_x) \notin DTH$. At the AS_j there are two cases as before.

(a) AS_j has not received the *aborted* message. As it has received the *committing* message it will check in the database for the transaction identifier. It cannot find the transaction

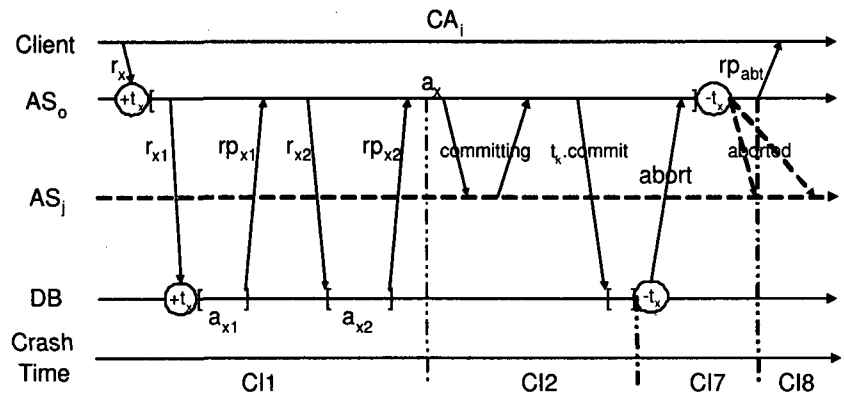


Figure 5.6: Possible crash intervals of the 1-1 algorithm in case of an abort at commit

identifier and then discard the changes. Thus, $AST(t_x) \notin ATH_j$. When CRM resubmits r_x, AS_j will reexecute as if it were a new request. As in case $CI5$ having two executions is correct as the first did not leave any effects in ATH_j, DTH and RPH_{CA_i}.

(b) AS_j has received the *aborted* message. Thus, it has discarded the changes of the previously received *committing* message and not added $AST(t_x)$ to ATH_j. From there, the reasoning is the same as in case $CI5b$.

CI8: Due to application semantics, $DBT(t_x) \notin DTH$. As above AS_j might have received the *aborted* message or checked for the database identifier in the database. In both cases, it does not append $AST(t_x)$ to ATH_j. All histories match.

5.4 1-1 Replication Algorithm for Relaxed State Consistency

In case a transaction commits, relaxed state consistency requires the same actions as full state consistency. Hence, we only consider the abort of a transaction in the following. With relaxed state consistency, even if a transaction aborts due to application semantics, it might change the state of the AS, but not the state of the database. Therefore, we have to replicate state changes performed by an AS transaction even in the abort case. As we have seen in the discussion above, abort is often database induced. In these cases the AS only is informed about the abort after it has taken

```
void abort_proc ()
  1. for each comp ∈ eu.COMP
  2.    set comp.state to current state
       of corresponding component;
  3. new aborted Message m3;
  4. m3.content = {eu};
  5. multicast m1 using uniform reliable delivery;
  6. wait until receive m3;
(a) primary: handle abort
```

```
void failover ()
  1. new Eu eu, new set COMP;
  2. in order of reception process each
     committing and aborted message m
  3.    eu = m.content;
  4.    if (m is committing message)
  5.       process eu as in the 1-1 algorithm;
       // see fig. 5.3 lines 4-12
  6.    if (m is aborted message)
  7.       for each comp ∈ eu.COMP
  8.          if (∃ c ∈ COMP and c == comp)
  9.             c.state = comp.state
 10.          else COMP = COMP ∪ {comp};
 11.       RR∪ = {(eu.req.rid, eu.resp)};
 12. for each comp ∈ COMP
 13.    create corresponding component;
 14.    set component's state to comp.state;
(b) failover
```

Figure 5.7: "1-1-relaxed" algorithm to support relaxed state consistency

place in the database, i.e., after the fact. Different to the commit case, this implies that state change propagation cannot always be performed before the transaction actually aborts but only afterwards.

Figure 5.7 shows the changes to the 1-1 algorithm of Figure 5.2 to support relaxed state consistency. When a transaction is aborted, the abort routine (Figure 5.7 (a)) sends an *aborted* message including the final state for each accessed component and the pair of the client request and the abort response. The *aborted* message is sent with uniform reliable delivery, and execution only continues when the primary receives its own aborted message. This means, that the user only receives the abort response when it is secured that the backups know about the abort and the corresponding state changes at the AS. The backup stores each aborted message in the FIFO queue together with other messages. Figure 5.7 (b) shows the modified failover. Both committing messages and aborted messages are processed in FIFO order to track the latest state of each component (lines 2-3). For a committing message, it will be processed as was the case for the full state consistency (lines 4-5). For an aborted message, all components affected will be recorded with its latest state and the pair of the client request and the corresponding abort response will be recorded in RR (lines 6-11). Finally, all necessary components are recreated (12-14).

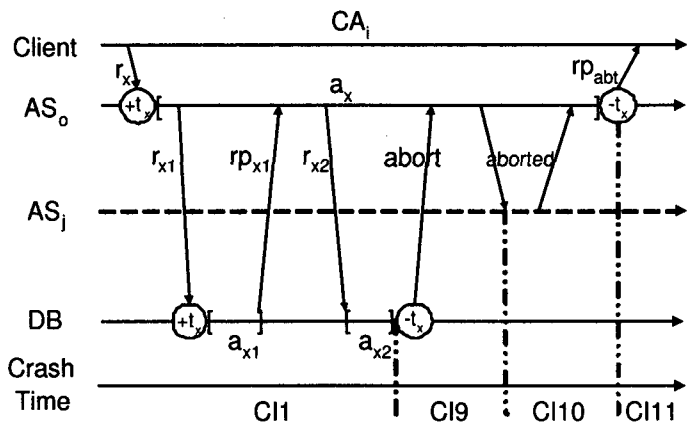


Figure 5.8: Possible crash intervals of the relaxed state consistency algorithm in case of an abort during execution

5.5 Correctness of Relaxed State Consistency Algorithm

The main difference compared to the full consistency algorithm is that the aborted message contains the state changes and is sent with uniform reliable delivery. Figure 5.8 shows an abort during execution. The only difference to the execution for full state consistency, shown in Figure 5.5, is that the abort message is guaranteed to have arrived at the backups before the abort response is returned to the client. Crash interval $CI1$ remains as before and is not further discussed. $CI9$ now starts with the abort at the database and ends just before the abort message is sent to the backups. $CI10$ starts with sending this message and ends just before sending the client abort response, and $CI11$ starts after sending this abort response.

CI9: The CRM receives a failure execution. As the DB transaction aborts, $DBT(t_x) \notin DTH$. AS_j does not receive the *abort* message. The behavior and the reasoning for correctness is the same as $CI5a$. When CRM resubmits r_x, AS_j will reexecute as if it were a new request as AS_j does not know anything about t_x. Although there are now two executions this is correct as the first execution did not leave any entries in ATH_j or DTH and no response to the client. The second execution might again lead to an abort or to a commit. At this time, all histories match.

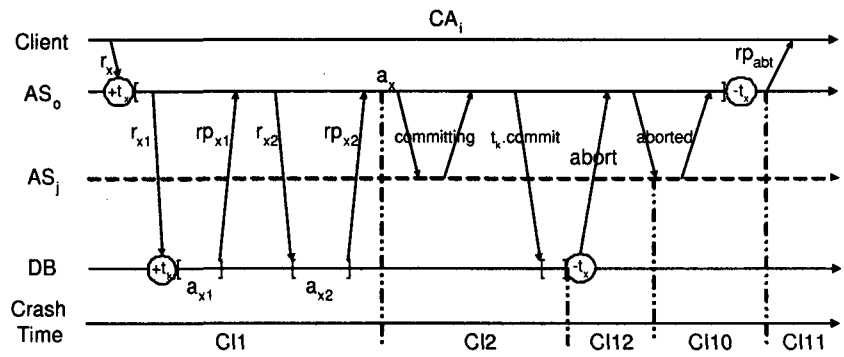


Figure 5.9: Possible crash intervals of the relaxed state consistency algorithm in case of an abort at commit

CI10: AS_j receives the *aborted* message. In this case, AS_j applies the changes contained in the abort message at failover, i.e., $AST(t_x) \in ATH$. With this ATH and DTH match. AS_j also puts the request/response pair into RR. When CRM resubmits r_x, the abort response rp_x is immediately returned. Now request and response histories match with each other and with ATH.

CI11: Request and response histories already match because the client has already received the response. AS_j is guaranteed to have received the *aborted* message and appended $AST(t_x)$ to ATH. Thus, request/response histories match with ATH. Finally, DTH and ATH match because it reflects the execution at the old primary.

Figure 5.9 shows the abort at commit time. $CI1$ and $CI2$ are as in the commit case. $CI12$ starts with the abort of the database transaction and ends before the backups receive the *aborted* message. The behavior in case the backups receive the *aborted* message is the same as before and discussed in $CI10$ and $CI11$.

C12: The behavior and reasoning is the same as $CI7a$. As AS_j has received the committing message it will check in the database but not find the transaction. Thus, upon resubmission of the request, a second execution occurs. This is correct, as the first execution at the old primary hasn't left any state changes in the system nor a response was returned.

Chapter 6

Advanced Algorithms for Advanced Execution Patterns

In this chapter, we describe advanced algorithms that are extended from the 1-1 algorithm for advanced patterns, namely the N-1 pattern and the 1-N pattern.

6.1 N-1 Pattern

In the N-1 algorithm, several client requests are associated with a single transaction. The basic idea of the N-1 algorithm is similar to the 1-1 algorithm. The primary AS propagates all state changes on session-related data performed by a transaction to backups at the commit time of the transaction. A backup only applies the state changes after it knows that the transaction has actually committed. Using this approach allows the new primary AS_j to match ATH_j and DTH after the old primary AS_o crashes. However, for the N-1 pattern, matching ATH_j and RH_{CA_i}/RPH_{CA_i} and matching RH_{CA_i} and RPH_{CA_i} of a client session CA_i are more complex than for the 1-1 pattern.

The main problem occurs if a client has already submitted a sequence of requests $r_1 \ldots r_k$ all belonging to a transaction t and has already received responses for $r_1 \ldots r_{k-1}$ when the primary crashes. As the transaction was still active, no transaction exists in DTH and the new primary AS_j also does not have any state changes. Thus, ATH_j does not match with RH_{CA_i} and RP_{CA_i}.

As a solution, we resubmit all requests $r_1 \ldots r_k$ to AS_j and not only the last request r_k to AS_j and execute them within a new transaction t'. The challenge is that the reexecution of r_1 to r_{k-1} should generate the same responses as their original execution at the old primary. Furthermore, these responses should not be seen by the client, as it has already received them during the original execution at the old primary. Receiving them twice would mean that RH_{CA_i} and RPH_{CA_i} don't match anymore.

If each response $rp_1 \ldots rp_{k-1}$ generated during the reexecution is the same as the corresponding response during the original execution, ATH_j eventually matches RH_{CA_i} and RPH_{CA_i}. If reexecution of one of the requests does not lead to the same response, we abort the transaction and return an abort as response to rp_k. Although the abort is not due to application semantics but due to the failure of the old primary, this guarantees that all matching requirements are fulfilled. We present two algorithms. The *N-1-best-effort* algorithm is simple and fast, but many transactions might be aborted because their reexecutions produce different responses. The *N-1-ordered* alternative achieves better transparency and a lower rate of aborted transactions at the price of higher overhead during normal processing.

Note that our algorithm is different from many implementations in current systems [78, 60, 56] that propagate state changes every time a response is returned to the client. In the above example, that would mean, state changes are propagated before rp_1 is returned, before rp_2 is returned etc. If the old primary now crashes before returning the response for r_k, AS_j would have the state changes triggered by $r_1 \ldots r_{k-1}$, and possibly also those of r_k. However, the database transaction still aborts completely. If AS_j applied these state changes and only the outstanding request r_k would be reexecuted, DTH would not match ATH_j.

In the following we first discuss full state consistency, and then discuss the changes needed to support relaxed state consistency.

6.1.1 N-1-best-effort

As mentioned in Section 4.2.2, the main difference between the N-1 pattern and the 1-1 pattern is that the client side controls the demarcation of transactions in the N-1 pattern. Hence, the main

changes compared to the 1-1 algorithm are at the client side. The client replication algorithm intercepts all requests, including begin, commit and abort requests. For each transaction, it keeps track of all component requests made so far and the corresponding responses. Only at commit time the server replication algorithm sends the primary state changes to the backups, not for each individual request. If the primary crashes while a transaction was active, the client algorithm resubmits all requests associated with the transaction to the new primary where they are executed within a new transaction. If reexecution leads to the same responses as the original execution, the new primary has equivalent actions to the actions performed at the old primary. Hence, reexecution was successful and failover is completely transparent. If it leads to different results, the replay was unsuccessful and the reexecuted transaction is aborted. The real client, having seen the old non-repeatable responses, is informed with a failure exception.

Our detailed algorithm description uses similar notations as the 1-1 algorithm. In the N-1-best effort algorithm, a single *CEU* object *ceu* at the CRM keeps track of the execution within the current transaction. It contains the transaction identifier $txid$ and all requests executed so far together with their responses (RR). The server maintains an EU object for each currently active transaction but does not need to keep track of request/response pairs. Additionally, the server use a set AT to record each aborted transaction.

The CRM (Figure 6.1) intercepts begin, invoke, commit and abort requests. For simplicity of description, we assume that the client submits requests in the correct order (begin/invoke/invoke.../commit). If a request to a component results in an abort, we expect the client to not continue with the transaction but submit a new begin transaction as next request.

Upon intercepting the begin request (Figure 6.1 (a)), the *ceu* object is initialized and the request is forwarded to the current primary until it is successfully executed. Upon a component request (Figure 6.1 (b)), the response from the primary is captured (lines 3-9). If the primary crashes before a response is received, we have to consider two cases. Firstly, the primary might have been in the middle of executing the request. Secondly, the request might have led to an application induced abort. The abort might have completed on the primary and the primary already informed the backups about this abort, but the primary crashed before returning the response to the client. In this case, the new primary is aware of this (unsuccessful transaction). Therefore, when the CRM receives

```
void begin ()
  1. while (true)
  2.   ceu.initialize();
  3.   ceu.txid = primary.begin();
  4.   if (∄ failure Exception) return;
  5.   else find a new primary;
```
(a) intercept transaction begin

```
Response invoke (Request req, Component comp)
  1. Generate req.rid;
  2. while (true)
  3.   Response resp =
  4.     primary.invoke(req, comp, ceu.txid);
  5.   if (resp == abort Exception)
  6.     throw abort Exception;
  7.   if (∄ failure Exception)
  8.     ceu.RR ∪ = {(req, comp, resp)};
  9.     return resp;
 10.   else
 11.     while (∃ failure Exception)
 12.       find a new primary;
 13.       if (primary.is_aborted(ceu.txid))
 14.         ceu.initialize();
 15.         throw abort Exception;
 16.       else
 17.         replay(ceu);
 18.         if (∃ replay failure)
 19.           ceu.initialize();
 20.           throw replay failure;
```
(b) intercept component request

```
void commit ()
  1. while (true)
  2.   primary.commit(ceu.txid);
  3.   if (∄ failure Exception)
  4.     ceu.initialize();
  5.     return;
  6.   else
  7.     while (∃ failure Exception)
  8.       find a new primary;
  9.       if (primary.is_committed(ceu.txid)) or
     (primary.is_aborted(ceu.txid))
 10.         ceu.initialize();
 11.         return;
 12.       else
 13.         replay(ceu);
 14.         if (∃ replay failure)
 15.           ceu.initialize();
 16.           throw replay failure;
```
(c) intercept transaction commit

```
void abort ()
  1. primary.abort(ceu.txid);
  2. throw abort Exception;
```
(d) intercept transaction abort

```
void replay (CEU ceu)
  1. ceu.txid = primary.begin();
  2. if (∃ failure Exception) throw failure Exception
  3. else
  4.   for each (oreq, ocomp, oresp) ∈ ceu.RR
  5.     Response nresp = primary.invoke (oreq,
     ocomp, ceu.txid);
  6.     if (∃ failure exception) throw failure Exception
  7.     else if (∃ abort exception) throw replay failure
  8.     else if (nresp != oresp)
  9.       primary.abort(ceu.txid);
 10.       throw replay failure;
```
(e) replay

Figure 6.1: N-1-best-effort at the client side

a failure exception (line 11), it checks at the new primary if the corresponding transaction had regularly aborted. If yes, the client algorithm simply returns the abort response, which is an abort exception (lines 13-15). Otherwise, a replay is initiated at the new primary (lines 17-20). Upon a commit request (Figure 6.1 (c)), if no crash happens, the termination is successful and returned to the user (line 2-5). If a crash occurred before the server returns from the commit, the transaction might have committed before the crash, aborted at commit time or aborted upon the crash. The *CRM* checks this at the new primary (lines 7-8). If the transaction committed or aborted at the

```
Response invoke (Request req, Component comp, TID txid)
  1. eu.COMP ∪ = {comp};
  2. Response resp = comp.invoke(rep)
  3. if (req is a client request and resp == abort exception)
  4.   abort_proc(eu);
  5. return resp;
(a) intercept request to component

void abort_proc (EU eu)
  1. AT ∪ = {eu.txid};
  2. new aborted Message m3;
  3. m3.content = {eu.txid};
  4. multicast m3 by reliable delivery;
(b) abort procedure

void abort (TID txid)
  1. TM.abort_Transaction(txid);
  2. abort_proc(eu);
(c) intercepts abort request

Bool is_committed (TID txid)
  1. if txid can be found in database return true
  2. else return false;
(d) check commit of transaction

Bool is_aborted (TID txid)
  1. if (∃ txid ∈ AT) return true;
  2. else return false;
(e) check abort of transaction

void failover ()
  1. ... // see Fig. 5.3 lines 1-12
  2. for each aborted message m with
     m.content == txid
  3.   AT∪ = {txid};
  4. ... // see Fig. 5.3 lines 16-18
f) failover at the new primary
```

Figure 6.2: N-1-best-effort at primary

time of commit, the request returns accordingly (lines 9-11). Otherwise, the transaction is replayed at the new primary (lines 13-16). When the client submits an abort request, it is simply forwarded and considered successful independently of whether a crash occurred or not (Figure 6.1 (d)). The replay (Figure 6.1 (e)) starts a new transaction at the new primary and resubmits each request of the old execution (lines 1-5). If one of these requests receives a different response than the original execution, the reexecuted transaction is aborted throwing a replay failure exception (lines 7-10). It is now up to the client to act upon this. Otherwise, reexecution has been successful and the algorithm continues with the request that was active at the time of the crash. Note that after the reexecution the state of the new primary (or the database) might not be exactly the same as the state of the old primary after the first execution. This does not really matter because only responses but not server state is visible to the client. Throughout the algorithm additional AS crashes reset the algorithm to the appropriate place.

At the server side, the N-1-best-effort algorithm is very similar to the 1-1 algorithm. We ignore the transaction begin and commit methods since they are the same as in the 1-1 algorithm. For a regular request (Figure 6.2 (a)), we only keep track of each component accessed by a request, but do not record the request/response pair as this is now done by the CRM. If the transaction aborts during request execution, an abort exception is thrown to the client as the response. Before that,

however, the abort procedure is called (Figure 6.2 (b)) which informs the backups about the abort, and stores the transaction identifier in the list of aborted transactions AT. The same procedure is called when the client requests an abort (Figure 6.2 (c)). The commit is the same as in the 1-1 case multicasting a committing message and a committed message. When the CRM checks if a transaction is committed, the *is_committed* routine (Figure 6.2 (d)) looks in the database for the $txid$ and returns the answer. Correspondingly, the *is_aborted* routine (Figure 6.2 (e)) looks in the set AT for the $txid$ and returns the answer. The failover at the new primary (Figure 6.2 (f)) is also similar to the 1-1 failover algorithm of Figure 5.3. However, instead of maintaining RR, AT must now be updated.

6.1.2 Correctness of N-1-best-effort

Correctness reasoning is similar to the proof of correctness of the 1-1 algorithm (see Section 5.3.1) for successfully completed transactions where the result was returned to the client without crash. The challenge lies in the case when the primary AS_o crashes while a client session CA_i has an active transaction t_x, i.e., the client has not yet received a commit or abort confirmation for t_x. Assume t_x involves $k+1$ client requests $r_x...r_{x+k}$. The primary AS_o might crash at any time point during the execution. We again discuss correctness by enumerating different crash intervals.

Commit case Figure 6.3 extends the execution in case of a commit for the N-1 pattern shown in Figure 4.3. In the figure, the client session CA_i submits a sequence of requests r_x to r_{x+k} within the transaction t_x on the replicated AS. In this example, the sub-request r_{y1} submitted by the request r_{x+1} and the sub-request r_{y2} submitted by the request r_{x+h} update the database, which make t_x be a DB update transaction. After the crash, we assume AS_j takes over as new primary and is up for sufficiently long to answer any outstanding requests and transactions. To prove the correctness, we check critical time points one by one. The first crash interval (CI1) ends just before the primary sends the response of the first request r_1, which is the transaction begin request in the N-1 pattern. If the primary crashes within $CI1$, the CRM resubmits the request r_1 to the new primary AS_j, and then sends all following requests to AS_j. This case is similar to the $CI1$ case of the 1-1 algorithm, where all requests are completely reexecuted on the new primary AS_j. We omit the discussion for

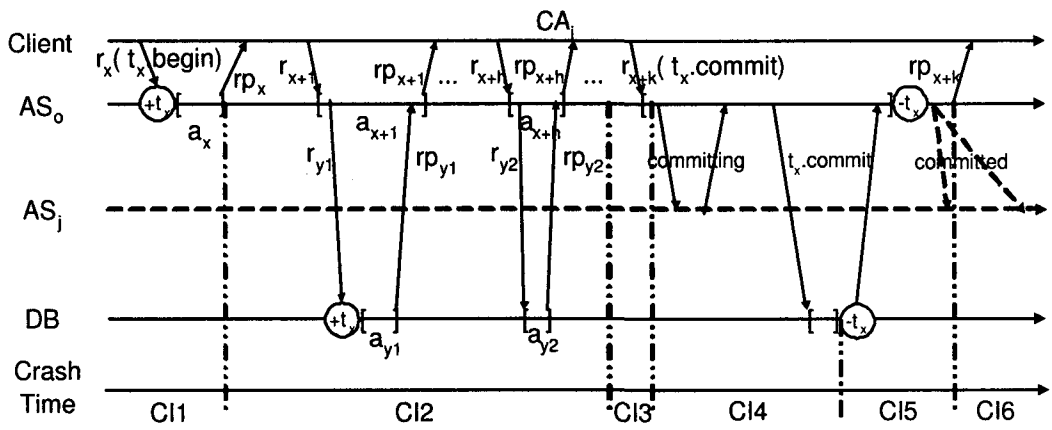


Figure 6.3: Possible crash intervals of the N-1 algorithm in case of a commit

this case. The second crash interval (CI2) starts when the response of the first request is sent to the client and ends just before the response of the request r_{x+k-1}, that is the last request before the commit request, is sent. The third crash interval (CI3) starts when the response of the request r_{x+k-1} is sent and ends just before the primary sends the committing message. That is, some time during interval $CI3$, the client submits the commit request r_{x+k}. As mentioned in Section 5.3.2, the uniform reliable delivery is modeled as a message/acknowledgement pair between primary and backup. The primary commits the transaction t_x only after it receives the logical confirmation that backups have already received the committing message of t_x. The fourth crash interval (CI4) starts after the primary has sent the committing message (i.e., the backups have the state changes) and ends just before the commit of the database transaction. The fifth crash interval (CI5) starts with the database commit (which adds this transaction to DTH) and ends just before the primary sends the response to the client. The sixth interval (CI6) starts after the response is sent (which allows the client to add it to RPH).

Let's now discuss what happens if the primary AS_o crashes while being in one of these crash intervals.

CI2: Without losing generality, we assume that the crash occurs during processing the request r_{x+h} $(0 \leq h < k)$. At this moment, the client session CA_i already received a sequence of responses

$rp_x ... rp_{x+h-1}$, which are contained by RPH_{CA_i}. The CRM receives a failure exception for the outstanding request r_{x+h}. AS_j has not received the *committing* message and has no knowledge about the transaction t_x. The database transaction, if it exists, is aborted upon the crash of AS_o. Thus, $AST(t_x) \notin ATH_j$ and $DBT(t_x) \notin DTH$. The CRM resubmits requests from r_x to r_{x+h} to AS_j, where each request is executed as a completely new request within a new transaction t'_x. The CRM checks whether or not each new response generated during the reexecution is the same as the corresponding original response of the same request. There are two situations:

(a) If all responses $rp_x \ldots rp_{x+h-1}$ generated during the reexecution of r_x to r_{x+h-1} are the same as the corresponding original responses, the CRM suppresses all of these duplicate responses. Then the response rp'_{x+h} is returned as the response to the outstanding request r_{x+h}. Then, AS_j continues executing requests from r_{x+h+1} to r_{x+k}, returning responses rp'_{x+h+1} to rp'_{x+k} to the client. Finally, RPH_{CA_i} contains $rp_x \ldots rp_{x+h-1}$ $rp'_{x+h} \ldots rp'_{x+k}$ that match requests $r_x \ldots r_{x+k}$ contained by RPH_{CA_i}. If transaction t'_k eventually commits, $AST(t'_x)$ is added to ATH_j, and $DBT(t'_x)$ is added to DTH if t'_x is a DB update transaction. Hence, all matching requirements are fulfilled. The t'_x also might be aborted due to application semantics. In this case, the last request associated with t'_k will have an abort response, $AST(t'_x) \notin ATH_j$ and $DBT(t'_x) \notin DTH$. All matching requirements are fulfilled again.

(b) If any response generated during the reexecution is different from the corresponding original response, e.g., the response rp'_{x+g} $(1 \leq g \leq x+h-1)$ of the request r_{x+g} is not equivalent to the original response rp_{x+g}, then the reexecution is stopped and the transaction t'_x is aborted. The abort response $rp_{ab_{t'_x}}$ is returned as the response for the outstanding request r_{x+h}. As a result, RH_{CA_i} contains $r_x \ldots r_{x+h}$ and RPH_{CA_i} contains $rp_x \ldots rp_{x+h-1} rp_{ab_{t'_x}}$, and $RH_{CA_i} \bowtie RPH_{CA_i}$. $AST(t'_x) \notin ATH_j$ and $DBT(t'_x) \notin DTH$. It is obvious that $RH_{CA_i} \bowtie RPH_{CA_i}$ and $ATH_j \bowtie DTH$. The last request r_{x+h} has an abort response of t'_x, which fulfills Condition 2a of the N-1 matching property (Definition 4.3.3). Thus $ATH_j \bowtie RH_{CA_i}/RPH_{CA_i}$. However, the

abort of t'_x is not caused by application semantics, but caused by an failure, and hence, fault-tolerance is not transparent.

CI3: This case is similar to the above case $CI2$, where $AST(t_x) \notin ATH_j$ and $DBT(t_x) \notin DTH$. The CRM already received the responses for requests r_x to r_{x+k-1}, and receives a failure exception for the last request r_{x+k}, which is the commit request of the transaction. As in $CI2$, the CRM has to start the resubmission from request r_x.

CI4: The CRM already received the responses for requests r_x to r_{x+k}, and receives a failure exception for the outstanding commit request r_{x+k}. AS_j has already received the *committing* message for t_x. If $eu.db = false$, then AS_j adds $AST(t_x)$ to ATH_j at failover. When CRM resubmits the request r_{x+k}, AS_j immediately returns the successful commit response. Correctness is given, as request, response and ATH match. If $eu.db = true$, AS_j knows that a DB update transaction was involved but aborted, since it cannot find the transaction identifier in the database. Thus, AS_j discards the content of the *committing* message. Therefore, neither ATH_j nor DTH have the transaction t_x. Then, the CRM resubmits the request starting from r_x as discussed in $CI2$.

CI5: The CRM receives a failure exception for the last request r_{x+k} and $DBT(t_x) \in DTH$. At AS_j, the *committing* message was received. At failover, AS_j detects that $DBT(t_x)$ has committed because it either has already received the *commit* message or it has looked for and found the transaction identifier in the database. Thus, $AST(t_x)$ is added to ATH_j, and $ATH_j \bowtie DTH$, and $ATH_j \bowtie RH_{CA_i}$. When the CRM resubmits r_{x+k}, AS_j immediately returns the successful commit response, and $ATH_j \bowtie RPH_{CA_i}$ and $RH_{CA_i} \bowtie RPH_{CA_i}$.

CI6: The client receives the response from the old primary, $RH_{CA_i} \bowtie RPH_{CA_i}$ and there is no resubmission. From the perspective of AS_j this is the same as $CI5$ and $AST(t_x)$ is added to ATH.

Aborts In the N-1 pattern, the transaction t_x can be aborted in three different cases. The first abort case is that an abort is caused during execution by the AS or the database. Figure 6.4 (a) shows an

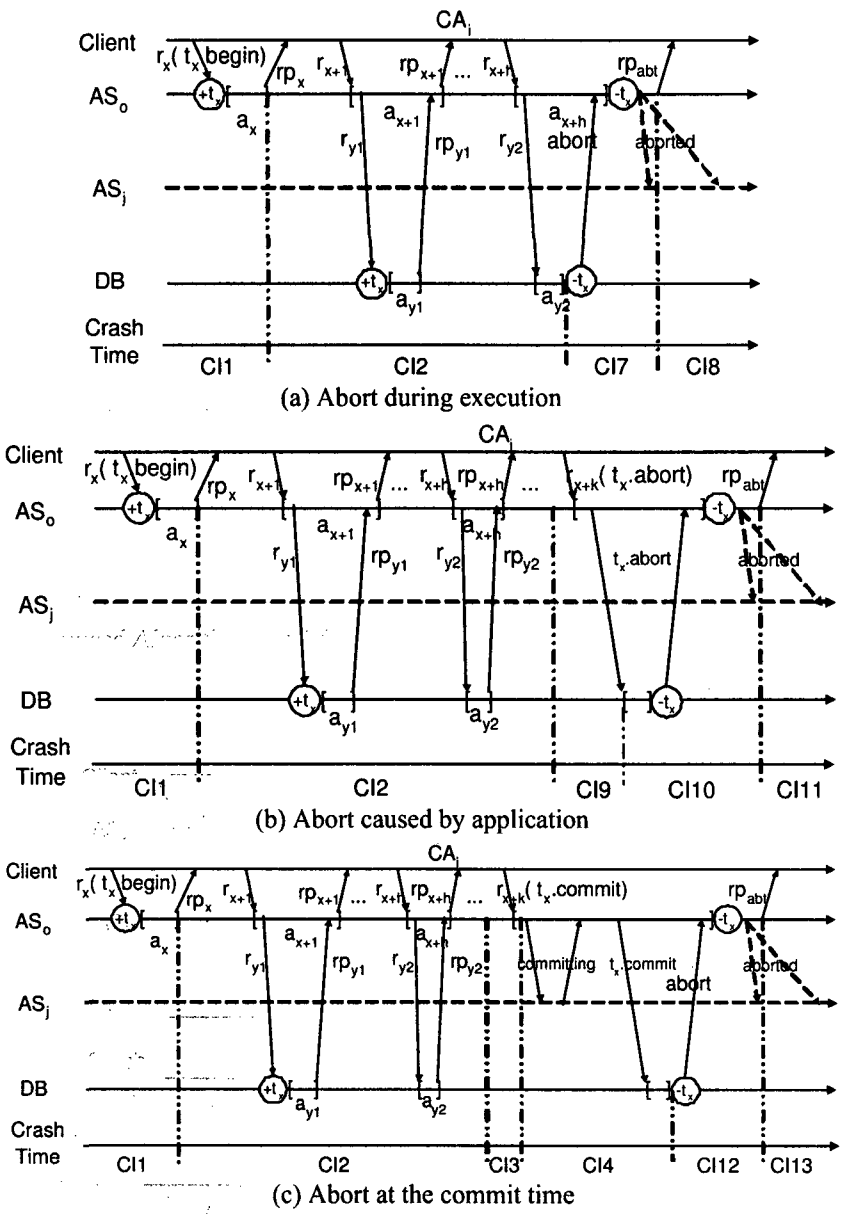


Figure 6.4: Possible crash intervals of the N-1 algorithm in case of a commit

example of the first case, where the database access action a_{y2} leads to an abort at the database. In this case, the first two crash intervals are exactly the same as the situation of the crash intervals

$CI1$ and $CI2$ of the commit case, and hence we omit the discussion. $CI7$ starts when the database aborts t_x. As in the 1-1 algorithm, the AS receives an abort response, aborts the AS transaction, sends an *aborted* message to backups using reliable delivery, and finally sends the abort response to the client. Crash interval $CI7$ ends just before the old primary sends the response to the client, and $CI8$ after sending this response.

CI7: The CRM receives a failure execution. As the DB transaction aborts, $DBT(t_x) \notin DTH$. For AS_j we can consider two cases.

(a) AS_j has not received the *aborted* message before the crash. In this case, AS_j has no knowledge of the abort, and hence $AST(t_x) \notin ATH_j$. The CRM receives the failure exception and then begins the replay from r_x. This is the same as the situation of the crash interval $CI2$ of the commit case.

(b) AS_j has received the *aborted* message, $AST(t_x) \notin ATH_j$. In this case, when CRM checks with the new primary AS_j if the transaction t_x is aborted or not, the answer is yes. Then, the CRM directly returns the abort response of t_x to the client as the response of the outstanding request r_{x+h} without resubmission. Request, response, ATH and DTH histories match with both AS and DB transactions aborted.

CI8: $DBT(t_x) \notin DTH$ as the DB transaction aborts. $AST(t_x) \notin ATH_j$, either because AS_j has received the *aborted* message or because it hasn't received any message at all. The CRM will not resubmit the outstanding request r_{x+h} since it already received the abort response. All histories match.

The second abort case is that an abort is caused by the application at the end of the execution. Figure 6.4 (b) shows an example of this case, where the client session submits an abort request as the last request r_{x+k} at the end of the execution. As before, the first two crash intervals are the same as the situation of the crash interval $CI1$ and $CI2$ of the commit case, and hence we ignore the discussion. Then, the client session submits the abort request to the AS. The AS receives the abort request, aborts the transaction t_k at the database first and then at the AS, sends an *aborted* message to backups using reliable delivery, and finally sends the abort response to the client. In regard to

crash intervals, we consider crash interval $CI9$ to start when the response to r_{x+k-1} is returned and to end before the database receives the abort request, $CI10$ to end before the old primary sends the abort response to the client, and $CI11$ after sending this response.

CI9: Although the database did not receive the abort request, the DB transaction is aborted due to the crash, and hence $DBT(t_k) \notin DTH$. The AS_j has no knowledge about the transaction, and hence $AST(t_k) \notin ATH_j$. The CRM receives a failure execution, but it will not resubmit the request. Instead, the CRM directly returns an abort response to the client as the response of the abort request. All histories match.

CI10: The DB transaction is aborted because it received the abort request and executed it, and hence $DBT(t_k) \notin DTH$. At the AS, there are two cases as before: AS_j either has already received the *aborted* message or not. In either case, $AST(t_k) \notin ATH_j$. The CRM receives a failure execution and then directly returns an abort response to the abort request without resubmission. All histories match.

CI11: $DBT(t_x) \notin DTH$ as the DB transaction aborts. $AST(t_x) \notin ATH_j$, either because AS_j has received the *aborted* message or because it hasn't received any message at all. The CRM will not resubmit the outstanding request r_{x+n} since it already received the abort response. All histories match.

The last abort case is that the database transaction aborts upon the commit request submitted by the AS as shown in Figure 6.4 (c). Crash intervals $CI1$, $CI2$, $CI3$, and $CI4$ are the same as in the commit case. $CI12$ starts with the abort at the database and ends just before the primary AS_o sends the abort response, and $CI13$ starts after sending this message.

CI12: The database transaction has aborted because of application semantics, thus, $DBT(t_x) \notin DTH$. The AS has two cases.

(a) AS_j did not receive the *aborted* message. As it has received the *committing* message it will check in the database for the transaction identifier. It will not find the identifier and discard the changes. Thus, $AST(t_x) \notin ATH_j$. The CRM receives the failure

exception and then begins the replay from r_x. This is the same as the situation of the crash interval $CI2$ of the commit case.

(b) AS_j has received the *aborted* message. Thus, it has discarded the changes of the previously received *committing* message and not added $AST(t_x)$ to ATH_j. From there, the reasoning is the same as in case $CI7b$.

CI13: Due to application semantics, $DBT(t_x) \notin DTH$. As above AS_j might have received the *aborted* message or checked for the database identifier in the database. In both cases, it does not append $AST(t_x)$ to ATH_j. The last commit request r_{x+n} receives the abort response. All histories match.

In summary, in all of above cases, all three matching properties are fulfilled, and hence the replication algorithm works correctly.

6.1.3 Relaxed State Consistency

The relaxed state consistency algorithm is a simple adjustment to the full state consistency algorithm. We only summarize the changes that have to be made to the N-1-best effort algorithm. At the server side, the multicast in the abort procedure (Figure 6.2 (b)) has to send the final state of all changed components. Furthermore, it must use uniform reliable delivery. Finally, the procedure only returns once the primary has received its own *aborted* message. For the client replication algorithm, also the abort changes (Figure 6.1 (d)). It has to implement similar steps as the commit (Figure 6.1 (c)). If it receives a failure exception upon an abort request, it has to contact the next primary. There it first checks whether the abort (including the relevant state changes at the AS) was successfully reported to the new primary. If yes, the abort was successful. Otherwise, we try to replay the transaction. If replay succeeds the transaction aborts due to application semantics as it did on the old primary. Also, the failover procedure at the new primary has to be slightly changed. It has to apply the state changes sent in *aborted* messages similar to the 1-1 algorithm for relaxed state consistency. If replay does not succeed, namely a duplicate response is not equal to the corresponding original response, we force the replay transaction to abort due to crash, and a corresponding

abort exception is returned to the client. We do not install the state changes performed by this abnormally aborted transaction in the current primary as is done for a normally aborted transaction in case of relaxed state consistency. We also do not propagate the state changes to backups. That is, this transaction is not contained in ATH. In fact, this violates Condition 2b of the N-1 matching property (Definition 4.3.3) that requires that for each request there is a corresponding $t \in ATH$. Instead this replayed transaction follows the rule of full state consistency. However, if we keep all changes of the replayed transaction, we would violate Condition 1 as it requires the response generated by this replayed transaction to be part of RPH_{CA_i}. But RPH_{CA_i} already contains the different response of the original transaction.

The other correctness reasoning is similar to the proof for relaxed state consistency of the 1-1 algorithm. Sending the aborted message with uniform reliable delivery guarantees that the client receives an abort response only when backups can have state changes performed by the corresponding aborted transaction. This, together with replaying transactions whose abort request was interrupted by a crash, makes sure that the state changes of aborted transaction are not lost, unless replay was unsuccessful.

6.1.4 Increasing the Chances for Exactly-Once

Reexecution might not succeed if non-determinism occurs which can happen because of database access. For example, assume before the primary crash, T_1 reads and updates x in the database, and returns a response to the client. Then the primary crashes before T_1 commits. At the new primary assume a transaction T_2 reads and updates x before T_1 resubmits its request. Hence, T_1's replay reads a different value of x than during the original execution. This might lead to a different response if the value of x affects the response. To avoid such behavior, we propose an alternative algorithm *N-1-ordered* that works for database systems that guarantee serializability through strict 2-phase locking. With N-1-ordered, the reexecution of all database access is performed in the same order as during the original execution. During normal processing, each database access is assigned a unique increasing identifier. Before the response for the request is returned, an *ordering* message with the identifiers of all access triggered by the request is multicast to the backups. If the request did not trigger any database access then no message needs to be sent. At failover, the new primary discards

an ordering message it received if the corresponding transaction committed before the crash, since client requests involved in this transaction will not be reexecuted. Otherwise, the request and the database access identifiers are recorded. When clients now resubmit their requests and reexecution starts at the new primary, each replayed database access must be executed according to its original order and new requests may not start until all resubmissions have completed. In the example above, when T_2's request is submitted before T_1 resubmits its request, it has to wait until T_1's request is reexecuted to guarantee that T_1 again reads the same data as in the original execution. In order to handle clients that do not replay (e.g., they crashed by themselves), there is a timeout of how long a request is blocked. If T_1 does not resubmit its request within a certain time, T_2's request (and other waiting requests) will execute to guarantee termination.

6.2 1-N Pattern

In the 1-N pattern, a client request triggers an AS action which is associated with an outer transaction. During the execution of this action sub-requests can trigger one or more nested inner transactions, e.g., in J2EE, if the method called is marked with the `RequiresNew` attribute. Inner transactions can have further nested inner transactions. As we mentioned before, we only consider relaxed state consistency given that outer and inner transactions might access the same session state.

In our model, for each AS transaction there is exactly one request that triggers the transaction and the one AS action associated with the request is the only AS action that is part of the AS transaction. In the following, given an AS transaction t, $at(t)$ indicates the action associated with the transaction, and $R(at(t))$ is the request that triggered that action $at(t)$.

6.2.1 Sub-requests and Nested Transactions

Let's have a closer look at an example execution and derive some terminology using this example. The example ignores the database transactions. Of course, every AS transaction can have a corresponding database transaction. In Figure 6.5, client request r_1 triggers action a_1 within outer AS transaction t_1. a_1 submits r_2 triggering action a_2 and inner transaction t_2, etc. The request order is $r_1 \ldots r_7$. Derived from Figure 6.5, we can model the execution of client request r_1 as a

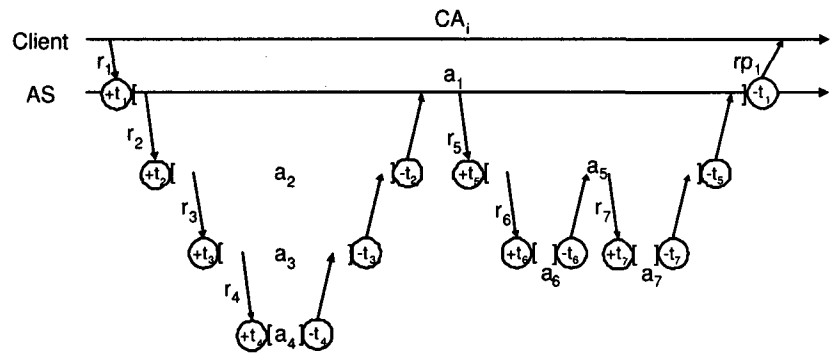
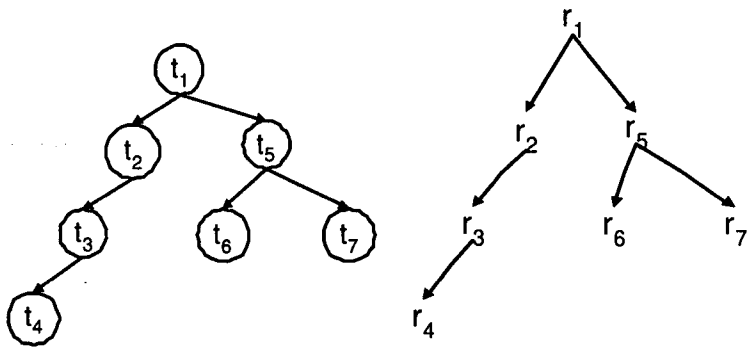


Figure 6.5: An example execution of the 1-N pattern



(a) Ttree of t_1 (b) Rtree of r_1

Figure 6.6: Ttree and Rtree

transaction tree $Ttree(r_1)$ as depicted in Figure 6.6 (a). The root of the tree is the outer transaction t_1. The children are the child transactions that are triggered by sub-requests submitted by the action a_1 associated with a_1. These child transactions can have further child transactions if they trigger transactions themselves. A child transaction always terminates before its parent transaction. Therefore, we obtain the termination order of transactions if we perform a post-order traversal of the transaction tree and denote this as $TSeq$. In our example, $TSeq(r_1) = t_4, t_3, t_2, t_6, t_7, t_5, t_1$.

In general, given a tree $Ttree(r_i)$ of client request r_i and rooted at outer transaction t_i, for any two transactions t_j and t_k in the tree, t_j is parent of t_k if there is an edge from t_j to t_k, and t_j is ancestor of t_k if there is a path from t_j to t_k. We refer to $ANCS(t_j)$ as the set of all ancestors of

t_j. Furthermore, for any transaction t_j being non-root node in $Ttree(r_i)$, t_j is inner transaction. Generally, a transaction t_j terminates before t_k if t_k is an ancestor of t_j (e.g., t_4 before t_2), or if both are siblings and $R(at(t_j))$ was submitted before $R(at(t_k))$ (e.g., t_6 before t_7), or they have a common ancestor with at least two children, and t_j is in the sub-tree rooted at the left child, and t_k is in the sub-tree rooted at the right child (e.g., t_3 before t_6). Note also that the client associated with root transaction t_i, i.e., $CL(GTX(t_i))$, is indirectly client of each other transaction t_j in $Ttree(r_i)$, thus, $CL(GTX(t_j)) = CL(GTX(t_i))$. Finally, for any transaction t_j in $Ttree(r_i)$ we denote as $cr(t_j) = r_i$, i.e., the client request that triggered the entire execution.

Complementary to $Ttree(r_i)$, we can build the request tree $Rtree(r_i)$ with client request r_i as root, and the sub-requests that triggered new transactions as descendants. Each node in this tree reflects the request that triggered the transaction at the same position in $Ttree(r_i)$. We define as $RPost(r_i)$ the post-order traversal of $Rtree(r_i)$ reflecting the order in which responses for the requests are returned. This is the same order as the termination order of the corresponding transactions. Furthermore, we define as $RPre(r_i)$ the pre-order traversal of $Rtree(r_i)$ which reflects the order in which requests are submitted. Finally, given a request r_j (client request or sub-request) of $Rtree(r_i)$, we denote as $cr(r_j) = r_i$, that is the client request that eventually led to r_j. Figure 6.6 (b) shows $Rtree(r_1)$ of our example. Then $RPre(r_1) = r_1, r_2, \ldots r_7$ and $RPost(r_1) = r_4, r_3, r_2, r_6, r_7, r_5, r_1$

With this, we define the *1-N matching property* for a client C_i as follows:

Definition 6.2.1. *$ATH \bowtie RH_{CA_i}/RPH_{CA_i}$ if the following holds:*

1. $\forall t \in ATH \wedge CL(GTX(t)) = C_i$:
   (a) *if t is an outer transaction, eventually* $R(at(t)) \in RH_{CA_i} \wedge RP(at(t)) \in RPH_{CA_i}$. *Furthermore, t aborts* $\Leftrightarrow RP(at(t)) = rp_{ab_t}$.
   (b) *if t is an inner transaction, eventually* $\forall t' \in ANCS(t), t' \in ATH$.
2. $\forall r \in RH_{CA_i}$: *eventually* $\exists t \in ATH \wedge r = R(at(t))$.
3. *Given* $t_1, t_2 \in ATH \wedge CL(GTX(t_1)) = CL(GTX(t_2))$:

(a) if $cr(t_1) \neq cr(t_2)$*:* $t_1 \prec t_2$ *in* $ATH \Leftrightarrow cr(t_1) \prec cr(t_2)$ *in* RH_{CA_i}.

(b) if $cr(t_1) = cr(t_2) = r_i$*: there exists one* $Rtree(r_i)$ *and* $t_1 \prec t_2$ *in* $ATH \Leftrightarrow R(at(t_1)) \prec R(at(t_2))$ *in* $RPost(r_i)$.

In the 1-N pattern, the relationship between client requests, responses and outer transaction is almost the same as in the 1-1 pattern. It differs from the 1-1 pattern in inner transactions. Condition 1a captures that each outer transaction in ATH has a matching client request and response, and generates an abort response only if it aborts. Condition 1b indicates that for each inner transaction, all ancestors must also be included in ATH as we assume relaxed state consistency. Condition 2 captures that each client request of client C_i has a matching outer transaction in ATH. Condition 3 captures the ordering property of transactions associated to a client C_i. Condition 3a indicates that transactions triggered by different client requests should be ordered in ATH in the same way the requests were submitted. Condition 3b indicates that if two transactions are associated with the same client request, then the requests that triggered these transactions must belong to the same request tree and the order of the two transactions in ATH reflects the nesting structure of the tree. Note that in the notation, any of t_1 or t_2 could be the outer transaction t_i.

6.2.2 1-N Algorithm Overview

The 1-N algorithm extends the 1-1 algorithm in order to handle outer and inner transactions and the relationship between them. For each individual transaction, the replication algorithm is the same as the 1-1 algorithm, where the primary propagates the state changes of the transaction to backups immediately before committing the database transaction using uniform reliable delivery and a backup only applies the state changes after it knows that the transaction has actually committed. Whereas, for the 1-N pattern, matching ATH and RH_{CA_i}/RPH_{CA_i} is more complex than for the 1-1 because of the inner transactions. We want to outline the main issues along three simple examples only considering commit cases.

Assume first a client request r_1 triggers an outer transaction t_1 and during its execution a subrequest r_2 starts an inner transaction t_2. If no crashes occur, then the primary first propagates the state changes of t_2 at commit time of t_2, and then the state changes of t_1 at commit time of t_1. When

the primary crashes there are now three main cases. If both t_1 and t_2 were still active then the new primary will not install state changes of any of the two transactions and no transactions related to r_1 are in DTH or ATH_j. Resubmission means complete reexecution. If both had committed, then no reexecution takes place and AS_j immediately returns the response. These cases are similar to the 1-1 pattern. The tricky case occurs if inner transaction t_2 has already committed but outer transaction t_1 was still active. In this case, the new primary can install the changes for t_2 but not for t_1. Both DTH and AS_j contain only partial changes for r_1. Request r_1 is resubmitted and the new primary has to reexecute starting a new transaction t_1'. If the execution is deterministic, t_1' will submit the very same request r_2 that initiated t_2 on the old primary. AS_j should not reexecute the request since t_2's state changes are already contained in the AS and the database. Thus, the new primary has to keep the request/response pair for t_2 which it received in the *committing* message. Then it can simply return the answer. After t_1' has completed, we can consider t_1' to be the parent of t_2 and the matching conditions are fulfilled. However, if execution is non-deterministic, the execution of t_1' might not trigger r_2 and then $r_2 = R(at(t_2)$ and $r_1 = R(at(t_1'))$ belong to two different $Rtree(r_1)$ which is a violation of the matching property. We consider t_2 a ghost transaction as it does not match the current execution. In this case, we abort the outer transaction t_1' providing a corresponding message to the application. Correctness is not provided. It is up to the application to handle ghost transaction t_2. If could do so with some compensating methods, it might apply when inner transactions commit and the outer transaction aborts during the standard execution.

The second example is a simple extension of the first to discuss sibling transactions. Assume that the execution of transaction t_1 does not only submit a sub-request r_2 to execute t_2 but after t_2 terminates, a further sub-request r_3 triggers transaction t_3. Crash situations where none or all of the transactions terminate, or where only t_2 terminates have already been covered with the first example. The additional case here is that both t_2 and t_3 committed while t_1 was still active. In this case, the reexecution of r_1 in transaction t_1' needs to resubmit both r_2 and r_3 and also in this proper order. Only then reexecution can be successful, and we have single request and transaction trees. If any of the two sub-requests are not regenerated or they are generated in different order, then the matching orders are violated. We abort t_1'.

The third example extends the first example to further discuss nesting transactions. Assume that

client request r_1 executes within outer transaction t_1. During execution, a sub-request r_2 triggers transaction t_2. During execution of t_2 a further sub-request r_4 triggers transaction t_4 which is now a child transaction of t_2. Crash situations where none or all of the transactions terminate are the same as in the first example. Now assume only t_4 committed before the crash but t_1 and t_2 were still active. Then ATH_j and DTH only contain t_4. During reexecution of r_1 in a new transaction t'_1, only if a request r_4 is resubmitted and the corresponding transaction is the first to be committed, the further execution of t'_1 can lead to a matching. In this case r_4 should not be reexecuted but the response of t_4 immediately returned as response to sub-request r_4. If both t_4 and t_2 were committed, during reexecution of r_1 in transaction t'_1, if t'_1 resubmits request r_2, the response of transaction t_2 should be immediately returned. As t_4 is nested within t_2, t'_1 is not expected to resubmit request r_4. The resubmission of r_2 implicitly includes the execution of t_2 and t_4. If t'_1 does not resubmit r_2, then both t_2 and t_4 become ghost transactions. Matching properties are violated, and we abort t'_1.

6.2.3 1-N Algorithm Details

In order to correctly reexecute an outer transaction that had already triggered inner transactions, we have to distinguish between client requests and sub-requests that trigger inner transactions, we need to know for an inner transaction which was the client request that triggered its outer transaction, and we need to know the request and transaction execution trees in order to know in which order requests were submitted and in which order transactions committed. For that, each request r_i has an attribute $r_i.cr$ pointing to request $cr(r_i)$ and an attribute $r_i.parent$ to indicate its parent request (if the parent request is not the client request itself). This information is part of the *committing* message.

Figure 6.7 shows the 1-N algorithm. We ignore the client part of the algorithm and the commit method and abort method of the algorithm since they are the same as those of the 1-1 algorithm for relaxed state consistency (see Figure 5.7).

Let's first have a look at failover in Figure 6.7 (c). The new primary analyzes the committing messages of all committed inner transaction triggered by the same client request in their receiving order and puts the requests leading to these inner transactions in a sequence queue (denoted as $RSeq$ in the algorithm). The receiving order of the inner transactions and the outer transaction of

```
TID begin (Request req)
  1. new EU eu;
  2. eu.txid = TM.begin_Transaction();
  3. eu.req = req;
  4. return eu.txid;
(a) intercepts a transaction's begin at primary

void failover ()
  1. new Eu eu, new set COMP;
  2. in order of reception process each
     committing and aborted message m
  3.   eu = m.content;
  4.   if m is a committing message and eu.db == true
     and ∄ committed message m' with m'.content ==
     eu.txid and eu.txid does not exist in database)
  5.         ignore committing message
  6.   else
  7.     for each comp ∈ eu.COMP
  8.       if (∃ c ∈ COMP and c == comp)
  9.         c.state = comp.state
 10.       else COMP ∪ = {comp};
 11.     RR∪ = {(eu.req.rid, eu.resp)};
 12.     find RSeq of eu.req.cr;
 13.     if (not found)
 14.       if (eu.req is a sub-request)
 15.         create RSeq for eu.req.cr;
 16.         append (eu.req, txid) to RSeq;
 17.     else
 18.       find the first item ereq in RSeq with
     ereq.parent == eu.req
 19.       if (found)
 20.         for each item dreq in RSeq from ereq to
     the end;
 21.           remove dreq from RSeq;
 22.           remove (dreq.rid, resp) from RR;
 23.       if (eu.req is a sub-request)
 24.         append (eu.req, txid) to the end of RSeq;
 25.       else
 26.         delete RSeq for eu.req
 27. for each comp ∈ COMP
 28.   create corresponding component;
 29.   set component's state to comp.state;
(c) failover at the new primary

Response invoke (Request req, Component comp, TID txid)
  1. New TID otid = null;
  2. if (req is a client request)
  3.   if (∃ (req.rid, resp) ∈ RR)
  4.     return resp ;
  5. else if (req is a sub-request and a new transaction is
     required)
  6.   find RSeq of req.cr;
  7.   if (found)
  8.     get the first item oreq from RSeq;
  9.     if (req == oreq)
 10.       find (oreq.rid, resp) from RR
 11.       remove the first item from RSeq;
 12.       return resp;
 13.     else if (isAncestor(req, oreq) == false)
 14.       throw abort exception for ghost transaction;
 15.   otid = txid;
 16.   TM.suspend_Transaction(txid);
 17.   txid = begin(req);
 18. find eu corresponding to txid;
 19. eu.COMP ∪ = {comp};
 20. Response resp = comp.invoke(req);
 21. if (req == eu.req)
 22.   eu.resp = resp;
 23.   if (resp == abort exception) abort(eu.txid);
 24.   else
 25.     if (RSeq ≠ Λ)
 26.       get the first item oreq from RSeq;
 27.     if (req is a client request and RSeq ≠ Λ)
     or (req is a sub-request and isAncestor(req, oreq))
 28.       resp = abort exception for ghost transactions;
 29.       abort(eu.txid);
 30.       break;
 31.     else
 32.       RR ∪ = {(req.rid, eu.resp)};
 33.   if (otid ≠ null) TM.resume_Transaction(otid);
 34. if (∃ abort exception for ghost transactions)
 35.   throw abort exception for ghost transactions;
 36. else return resp;
(b) intercepts a request at primary

Bool isAncestor (Request req1, Request req2)
  1. Request treq = req2;
  2. do
  3.   if (req1 == treq.parent) return true;
  4.   else treq = treq.parent;
  5. while (treq is a client request);
  6. return false;
(d) check the ancestor request of a request
```

Figure 6.7: "1-N" algorithm

a client request r_i is the post-order traversal of $Ttree(r_i)$. When we receive for a client request the first inner transaction t_i triggered by sub-request r_i, $RSeq$ for this client request is created and r_i

is added (lines 14-16). When we receive inner transaction t_i triggered by sub-request r_i and $RSeq$ already exists, we first check whether t_i had any nested child transactions (line 18). These nested transactions terminated before t_i, and thus, the sub-requests leading to them have already been added to $RSeq$. If such requests exist, they are removed from $RSeq$ and RR (lines 20-22). This is correct as the transaction associated with their parent request has committed, and any potential replay will directly get the response for the parent request and they will not be further replayed. Then r_i itself is added to $RSeq$ because at replay time this request should be checked (lines 23-24). If the same request is not made in proper order during replay, ghost transactions occur. If r_i is a client request, $RSeq$ for r_i can be deleted since all transactions related to r_i have now successfully terminated (lines 25-26). After this process, in $RSeq$, the sequence of sub-requests triggered by the original execution of the client request is sorted according to the order in which requests were submitted originally. However, the sequence might not be complete. If two transactions are siblings and both committed, their requests are included in $RSeq$ according to the commit order. If a transaction is included in $RSeq$ then none of its descendant transactions is included in $RSeq$. For instance, in the nesting example of the previous section 6.2.2, if t_2 and t_4 committed but not t_1, then only t_2 is included in $RSeq$. However, it might be that a transaction is included but its parent transaction is not included. In our example in the previous section where we had the nesting of t_1, t_2 and t_4, this occurs if t_4 committed but not t_2 and t_1. Then, only r_4 is included in $RSeq$.

Figure 6.7 (b) shows the algorithm on the primary. For a client request, we process it as the 1-1 algorithm (lines 2-4). If it is a duplicated request, its response is returned immediately without reexecution. For a sub-request, if it is detected that a new transaction will be triggered, we first check if the sub-request is triggered by a replayed client request (line 6). If yes, its client request should have $RSeq$ that is created at the time of failover. If $RSeq$ is found, the sub-request is compared with the current first request in $RSeq$ (lines 7-9). If they are the same request, namely, both requests calling the same method with same parameter values, the sub-request is regarded as the request that triggered the inner transaction that already committed at the old primary. In this case, the execution of the sub-request is suppressed, the current first item is pushed out from $RSeq$, and the replicated response is directly returned (lines 10-12). If they are not the same, we should check if the sub-request is an ancestor request of the current first request in $RSeq$ (it is checked by

the function `isAncestor` defined in Figure 6.7 (d)). If not, the corresponding inner transaction is regarded as a ghost transaction and an abort exception is thrown (lines 13-14). Otherwise, it is the case that an inner transaction is included in $RSeq$ but its parent transaction is not included. In this case, we can safely reexecute the sub-request, waiting for the possible resubmission of the first request in $RSeq$ during the reexecution of the sub-request. If $RSeq$ is not found, it means that the sub-request is not triggered by a replayed request and can be safely executed. In both cases, the sub-request can safely create a new inner transaction, suspending the parent transaction (lines 16-18). Then the sub-request is processed as in the 1-1 algorithm (lines 19-21). When execution of a leading request of a transaction is finished (it might be a client request or a sub-request and hence the transaction might be an outer or inner transaction) (line 22), $RSeq$ has to be checked again since there exists the possibility that the replay of the current request does not replay all sub-requests of all committed inner transactions of its original execution. Ghost transactions exist in two cases: (1) for a replayed client request, $RSeq$ is not empty, (2) for a replayed sub-request, it still has children requests in $RSeq$. In both cases the current transaction has to be aborted (lines 25-30). Otherwise, RR keeps request/response pairs for each committed transaction (lines 32). Then, if there is a suspended parent transaction, it will be resumed to be executed (line 33). In this case, TM holds the current transaction in a queue waiting for the termination command. Finally, if the current request gets an abort exception, the abort is thrown to its caller without returning the real response (lines 34-35). This guarantees in case that ghost transaction exists, the outer transactions will eventually be aborted. Otherwise, the real response will be returned. When the TM is notified that the request is returned successfully, it commits the corresponding transaction.

6.2.4 Correctness Discussion

Correctness reasoning of the 1-N algorithm is similar to the proof of correctness of the 1-1 and N-1 algorithm (see Section 5.3.1). The major challenge is to check the situation where the primary crashes when some inner transactions have become successful update transactions. For the sake of simplification and avoiding long-winded repetition, the section uses two examples to show how the 1-N algorithm guarantees the correctness for an outer transaction that has two sibling inner transactions and an outer transaction that has nested inner transactions. Other cases can be derived

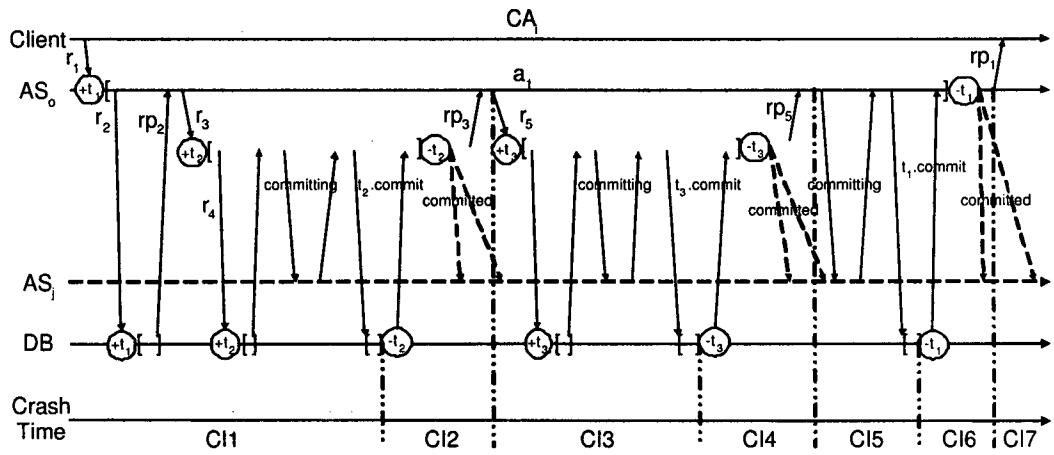


Figure 6.8: Possible crash intervals of the 1-N algorithm for sibling inner transactions in case of a commit

from there.

Figure 6.8 shows an example where an out transaction has two sibling inner transactions. In the figure, client action CA_i submits request r_1 to the primary AS_o. If no crash occurs, r_1 is executed as the action a_1 within the transaction t_1. The sub-request r_2 is a database update request. The sub-request r_3 of a_1 triggers an inner transaction t_2. Within t_2, a further sub-request t_4 updates the database. and then t_2 commits. Then, r_3's response is returned to a_1, and t_1 resumes. Then, a_1 submits a sibling sub-request r_5 triggering transaction t_3. Finally t_3 and then t_1 eventually commits and the response rp_1 is returned. In this scenario, the first crash interval $CI1$ ends before the inner transaction t_2 commits at the database. The second crash interval $CI2$ starts after the inner transaction t_2 commits at the database, and ends just before sending the sub-request r_5. Then, the third crash interval $CI3$ ends before the second inner transaction t_3 commits at the database. The fourth crash interval $CI4$ ends before the primary sends the committing message of transaction t_1 to backups. The fifth crash interval $CI5$ ends before t_1 commits at the database. Then the sixth crash interval $CI6$ ends just before sending the response rp_1 to the client request r_1. Finally the seventh crash interval $CI7$ ends after sending the response. Again, let's analyze all crash intervals one after another.

CI1: No matter whether AS_j received the state changes of the transaction t_2, AS_j knows that t_2 has aborted at the database. Hence, $AST(t_2) \notin ATH_j$. It matches that $DBT(t_2) \notin DTH$. The CRM receives a failure exception for the outstanding request r_1, and then resubmits r_1 to AS_j. AS_j does not have $RSeq$ for the client request r_1. AS_j executes r_1 as a completely new request. Finally, if AS_j can be up for sufficiently long to handle r_1's execution, all matching requirements will be fulfilled as a normal execution.

CI2: The CRM receives a failure exception. t_1 has aborted due to the crash and hence $DBT(t_1) \notin DTH$. t_2 has committed and hence $DBT(t_2) \in DTH$. At AS_j, the *committing* message of t_2 was received. At failover, AS_j can detect the commit of $DBT(t_2)$, and thus $AST(t_2)$ is added to ATH_j. AS_j has the $RSeq$ for the client request r_1. The $RSeq$ contains only the sub-request r_3 that triggers the committed inner transaction t_2. Then, the CRM resubmits r_1 to AS_j. AS_j executes r_1 in a new transaction t'_1. The first sub-request r' of the reexecution that will trigger a new transaction has to be compared with r_3, which is the first item in $RSeq$.

(a) If r' is equal to r_3, r' is suppressed, and the response rp_3 of r_3 that is contained by the committing message is returned as the response of r'. Then, the remaining part of the reexecution of r_1 continues as the execution of a new request. When t'_1 terminates, there are $AST(t'_1) \in ATH_j$ and possibly $DBT(t'_1) \in DTH$. There are also $AST(t_2) \in ATH_j$ and $DBT(t_2) \in DTH$. We can consider that t'_1 and t_2 are part of the same $Ttree(r_1)$ and $AST(t_2) \prec AST(t'_1)$ in ATH_j. There is also only one $Rtree(r_1)$ and $r' \prec r_1$ in $RPost(r_1)$. All matching properties are fulfilled.

(b) If r' is not equal to r_3, t_2 becomes a ghost transaction. Then, t'_1 is aborted and the aborted response $rp_{ab_{t'_1}}$ is returned to the client as the response of r_1. As a result, ATH_j contains $AST(t_2)$ and DTH contains $DBT(t_2)$, and ATH_j also contains $AST(t'_1)$. However, t'_1 and t_2 do not belong to the same $Ttree(r_1)$ and there is more than one $Rtree(r_1)$. Conditions 1b and 3 of the 1-N matching property are violated. The request $r_1 \in RH_{CA_i}$ has a matching response $rp_{ab_{t'_1}} \in RPH_{CA_i}$. This could be regarded as the satisfaction of Conditions 1a and 2 of the 1-N matching property.

(c) if the reexecution does not submit any sub-request to trigger an inner transaction, t_2

becomes a ghost transaction. It will be detected by our algorithm at the end of the reexecution of r_1 since $RSeq$ is not empty but still contains r_3. Then t'_1 is aborted and the abort response is returned to the client side. The same violations occur as above.

CI3: The CRM receives a failure exception for the outstanding request r_1. $AST(t_1) \notin ATH_j$ and $DBT(t_1) \notin DTH$ due to the crash. Since the crash occurs before the inner transaction t_3 eventually commits at the database, both ATH_j and DTH do not contain t_3. However, both of them contain t_2 since t_2 has committed at this moment. When the CRM resubmits r_1 to AS_j, the reexecution is the same as $CI2$.

CI4: In this case, ATH_j and DTH eventually contain both t_2 and t_3, since AS_j knows that both t_2 and t_3 already committed at the database. AS_j has the $RSeq$ for the client request r_1. The $RSeq$ contains two sub-requests $r_3 r_5$ that are leading request for t_2 and t_3. The CRM receives a failure exception and resubmits r_1 to AS_j. When replaying r_1 in a new transaction t'_1, the first two sub-requests that trigger new transactions have to be compared with r_3 and r_5 in the proper order. If both are equal, both sub-requests are suppressed, correctness reasoning is as in $CI2$a. Otherwise, t_2 and/or t_3 becomes ghost transactions, and correctness reasoning is similar to the case of $CI2$b.

CI5: AS_j received the committing message of t_1, but can detect that t_1 did not commit at the database. Hence, $AST(t_1) \notin ATH_j$ and $DBT(t_1) \notin DTH$. From there, this case is the same as $CI4$.

CI6: AS_j received the committing message of t_1, and can detect that t_1 already committed at the database. Hence, $AST(t_1) \in ATH_j$ and $DBT(t_1) \in DTH$. Also, $AST(t_2), AST(t_3) \in ATH_j$ and $DBT(t_2), DBT(t_3) \in DTH$. Since the outer transaction t_1 is already committed, $RSeq$ for the client request r_1 is deleted at failover. Then, when the CRM resubmits r_1 to AS_j, the original response of r_1 that is contained by the committing message of t_1 is directly returned without reexecution. As a result, all histories match.

CI7: From the perspective of AS_j this is the same as $CI6$. The only difference is that the client receives the response from the old primary. Thus, there is no resubmission.

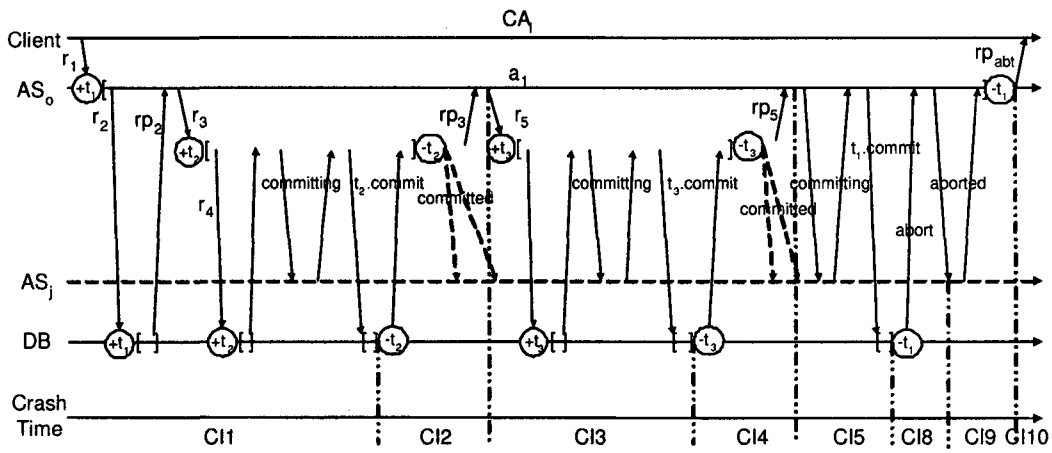


Figure 6.9: Possible crash intervals in case of an abort of an outer transaction

Figure 6.9 shows an example of an abort case, where the database transaction aborts upon the commit request. In this case, the first five crash intervals are the same as $CI1$ to $CI5$ of the commit case, and the different intervals are $CI8$, which starts after the database aborts the transaction t_1 and ends before the aborted message is received by AS_j, $CI9$, which ends before the abort response is returned to the client, and $CI10$, which ends after the abort response is returned.

CI8: This case is similar to $CI5$. AS_j received the committing message of t_1, but can detect that t_1 already aborted at the database. The difference between $CI5$ and $CI8$ is that in the former case the abort is caused by the crash and in the latter case the abort is caused by the application semantics. However, the new primary cannot distinguish. Correctness reasoning is as in $CI5$.

CI9: The AS_j received the aborted message of t_1, and can apply the state changes of t_1. Then, t_2, t_3 are contained by both ATH_j and DTH, and t_1 is only contained by ATH_j. AS_j deletes $RSeq$ for the client request r_1 since it received the aborted message of t_1. When the CRM resubmits the request r_1 to AS_j, the abort response of t_1 is returned as the response. All matching requirements are fulfilled.

CI10: This case is similar to $CI9$ except that the abort response of t_1 is already returned. Hence, the CRM will not resubmit r_1 to AS_j. Instead, it directly returns the abort response to the

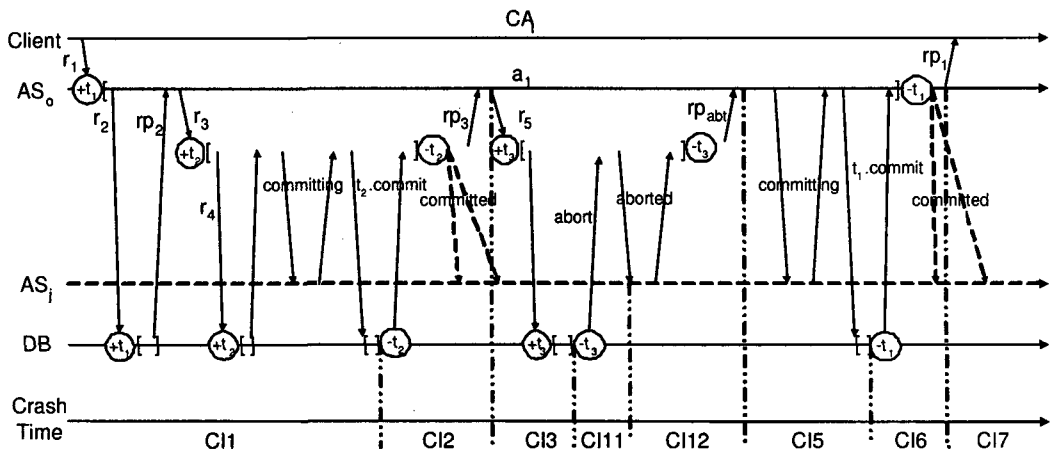


Figure 6.10: Possible crash intervals in case of an abort of an inner transaction

client.

The primary might also crash at other time points, the correctness reasoning is similar to the above case. Generally, if the new primary AS_j has already received the aborted message of the outer transaction, it will apply the state changes of the aborted outer transaction, and return the abort response to the client upon resubmission. Otherwise, the reexecution is similar to the commit case.

An inner transaction might be aborted, too. Figure 6.10 shows an example of the abort of the inner transaction t_3 in case of relaxed state consistency. Since each individual transaction can independently commit or abort, the outer transaction t_1 can commit after the abort of the inner transaction t_3. Most crash intervals in this case are similar to the commit case. The different crash intervals are $CI11$, which starts after the transaction t_3 is aborted at the database and ends before AS_j receives the aborted message, and $CI12$, which starts after the aborted message is received by AS_j and ends before the committing message of t_1 is propagated.

CI11: AS_j did not receive any information about t_3. After the crash, $AST(t_2) \in ATH_j$ and $DBT(t_2) \in DTH$. Then, the reexecution of r_1 is similar to $CI2$ of the commit case.

CI12: The AS_j received the aborted message of t_3, and can apply the state changes of t_3. Then,

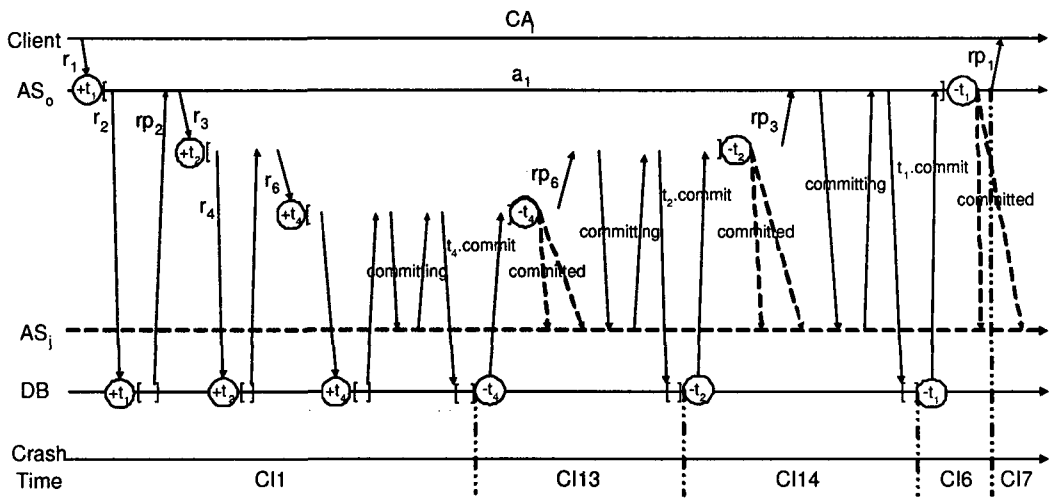


Figure 6.11: Possible crash intervals of the 1-N algorithm for nested inner transactions in case of a commit

$AST(t_2) \in ATH_j$ and $DBT(t_2) \in DTH$ and $AST(t_3) \in ATH_j$. Both t_2 and t_3 are successful update transactions. $RSeq$ of r_1 contains $r_3 r_5$ for t_2 and t_3. When the CRM resubmits the request r_1 to AS_j, the reexecution should check if t_2 and t_3 are ghost transactions and take the corresponding actions as $CI4$.

When an inner transaction is aborted, the outer transaction might be aborted as well. Correctness reasoning for this case is omitted since it is similar to previous cases.

Figure 6.11 shows an example where an out transaction has two nested transactions. In the figure, client action CA_i submits request r_1 to the primary AS_o. The client request r_1 is executing as an action a_1 within the transaction t_1. The action a_1 submits the sub-request r_3 to trigger an inner transaction t_2. Within t_2, a further sub-request r_6 triggers a nested inner transaction t_4. The committing order of these transactions is $t_4 t_2 t_1$. In this scenario, the first crash interval $CI1$ ends before the inner transaction t_4 commits at the database. The second crash interval $CI13$ starts after the inner transaction t_4 commits at the database, and ends just before the inner transaction t_2 commits at the database. The third crash interval $CI14$ starts after the inner transaction t_2 commits, and ends before the outer transaction t_1 commits at the database. Then the crash intervals $CI6$ and $CI7$ occur after the commit of the outer transaction. CI, $C6$ and $C7$ are the same as the similar

crash intervals of Figure 6.8. Hence, we only look at $CI13$ and $C14$.

CI13: The CRM receives a failure exception. t_1 and t_2 have aborted due to the crash. t_4 has committed and hence $DBT(t_4) \in DTH$. At AS_j, the *committing* message of t_4 was received. At failover, AS_j can detect the commit of $DBT(t_4)$, and thus $AST(t_4)$ is added to ATH_j. AS_j has the $RSeq$ for the client request r_1. The $RSeq$ contains only the sub-request r_6 that triggers the committed inner transaction t_4. Then, the CRM resubmits r_1 to AS_j. AS_j executes r_1 in a new transaction t_1'. The first sub-request r' of the reexecution that will trigger a new transaction has to be compared with r_6, which is the first item in $RSeq$.

(a) If r' is equal to r_6, r' is suppressed, and the response rp_6 of r_6 that is contained by the committing message of the transaction t_4 is returned as the response of r'. Then, the remaining part of the reexecution of r_1 continues as the execution of a new request. In this case, the reexecution of r_1 is in fact different from the original execution, since in the reexecution, r' is the direct sub-request of r_1 but in the original execution r_6 is a sub-request of the sub-request r_3, which is skipped in the reexecution. However, this is still a correct execution since r' is generated by the normal execution of r_1 and the response of r_6 should be the same as the response of r' since r' is equal to r_6. We can consider one $Rtree(r_1)$ that so far has pre-order $RPre(r_1) = r_1 r_6$ with corresponding transactions t_1' and t_4. The remaining execution completes the tree and all matching is provided.

(b) If r' is not equal to r_6, we check if r' is an ancestor request of r_6, namely check if r' is equal to r_3. If yes, r' is executed triggering a new transaction t_2'. If the execution of r' submits a sub-request r'' that will trigger a new nested inner transaction, r'' is compared with r_6 again. If r'' is equal to r_6, then r'' is suppressed and the response rp_6 is returned as the response of r''. Then, the remaining part of the execution of r' continues. We have now one $Rtree(r_1)$ that so far has pre-order $RPre(r_1) = r_1 r' r_6$ with corresponding transactions $t_1' t_2' t_4$. The remaining execution completes the tree and matching is provided. If r'' is not equal to r_6, or the execution of r' does not submit r'', or r' is not equal to r_3, t_4 becomes a ghost transaction. Then, t_1' is aborted and

the abort response $rp_{ab_{t'_1}}$ is returned to the client as the response of r_1. As a result, $AST(t_4) \in ATH_j$ and $DBT(t_4) \in DTH$ and $AST(t'_1) \in ATH_j$. The request r_1 has a matching response $rp_{ab_{t'_1}}$. The main violation lies in Conditions 1b and 3.

(c) if the reexecution does not submit any sub-request to trigger an inner transaction, t_4 becomes a ghost transaction. It will be detected by our algorithm at the end of the reexecution of r_1 since $RSeq$ is not empty but still contains r_6. Then t'_1 is aborted and the abort response is returned to the client side. Violations are as above.

CI14: The CRM receives a failure exception. t_1 has aborted due to the crash. t_2 and t_4 have committed and hence $DBT(t_4), DBT(t_2) \in DTH$. At AS_j, the *committing* messages of t_4 and t_2 was received. At failover, AS_j can detect the commit of $DBT(t_4)$ and $DBT(t_2)$, and thus $AST(t_4), AST(t_2) \in ATH_j$. AS_j has the $RSeq$ for the client request r_1. The $RSeq$ contains the only sub-request r_3 that triggers the committed inner transaction t_2. The sub-request r_6 is already removed from $RSeq$ since its parent transaction has committed. Then, the CRM resubmits r_1 to AS_j. AS_j executes r_1 in a new transaction t'_1. The first sub-request r' of the reexecution that will trigger a new transaction has to be compared with r_3, which is the first item in $RSeq$. This comparison is similar to $CI2$.

In summary, the 1-N algorithm guarantees correctness in some but not all cases. The problems are ghost transactions that do not match any proper tree perceived by the client. Since we only consider relaxed state consistency, the abort case for nested inner transactions is similar to the commit case and omitted here.

6.2.5 Undo Ghost Transactions

An alternative solution could be used if compensating transactions exist. Recall that if the 1-N pattern is used to chop a long execution into small pieces, compensating transactions are often provided by programmers. In this case, in the example above, when the discrepancy between an old request and a new request is detected, AS_j first executes the compensating transaction for the ghost transaction, and then continues execution with the new request. Please note that compensation transactions

have to be called in the reverse order of their commits. Compensating transactions lead to committing/committed messages as any other transactions. The effect is that a compensated transaction appears as if it had never been executed. However, if there exist some transactions that read data changed by the compensated transaction on the database, these transactions have to be undone as well, leading to "cascading compensation". The cascading compensation disseminates the ghost transaction problem from one client to other clients, and hence has to be handled carefully. This same problem already occurs during normal processing when committed transactions are undone by using compensating transactions. However, such mechanism appears complex. In particular at the implementation level it requires the replication algorithm have access to compensating transactions, which is often not feasible.

Chapter 7

Miscellaneous Extensions of ADAPT-SIB

In this chapter, we discuss miscellaneous extensions to our ADAPT-SIB replication tool. These extensions do not change the algorithm itself, but propose a couple of ways to make ADAPT-SIB adaptable to wider use cases where our assumptions for the algorithm might not hold, or the system has a more complicated architecture. These extensions handle different failover strategies, recovery of crashed replicas, request execution without transaction boundary, access of more than one database using 2PC, and crash of clients or the backend database.

7.1 Different Failover Strategies

An important parameter of fault tolerance is *failover time*, i.e., the period from the time point the backup detects the primary's crash to the completion of failover. In general, the new primary has to have the latest state of each component before any call to this component can be made. In the algorithmic descriptions of the last sections, the new primary creates all necessary components at the time of failover and installs for each the latest successful change. We call this strategy *Install-When-Failover*. However, creating components and setting their states can be very time consuming. Since the components are not accessible until failover is completed, clients might be blocked for a long time. Hence, although the crash exception is not exposed to clients, transparency might be lost since the system might seem to be frozen to the client.

To solve this problem, we propose two further restore strategies. In the *Install-Immediately* strategy, each backup creates a component when it receives the first message related to this component, and it knows that the corresponding transaction succeeded. The state of the component is refreshed immediately each time the backup receives a committing/committed message pair (or abort message in case of weak consistency) that refers to this component. With this strategy, each backup has a considerably higher load during normal processing when no crash occurs since not only the last state change but all state changes on a component are restored. However, during failover, the new primary only needs to consider components for which it had received a committing but no commit/abort message before the crash. For those, it has to check in the database whether the state changes recorded in the committing message should be installed. With this, failover can be very fast. While the *Install-Immediately* strategy speeds up failover by doing the necessary updates before a crash occurs, our last strategy improves on the failover time by delaying the necessary updates to when they are actually needed. We refer to it as *Install-After-Failover*. During normal processing, a backup simply queues all messages from the primary as with *Install-When-Failover*. At the time of failover, the new primary parses through the messages and only checks which components need to be restored (created and a final state installed). Then, it immediately allows client requests. Now, when a client submits a request to a specific component, if the component needs to be restored, the new primary recreates the component and installs the up-to-date state as found in the last relevant message. This strategy slows down only those requests that are the first to access a component that needs to be restored. However, the failover time during which all client requests are blocked, is very short. Additionally, when the system runs for a long time, it might occur that many components that were replicated are actually never reused again. Using the *Install-After-Failover* strategy, these components will never be created at the new primary but only those components are recreated that are really needed.

Note that the AS components we consider have usually a limited life-time and are then deleted, because we mainly consider components that maintain all relevant information in regard to a user session. The *Install-Immediately* strategy will create and delete such components on the backups basically in real-time. In contrast, the other two strategies only create components at or after failover. In order to avoid creating components at or after failover that were deleted, the replication algorithm

during normal processing can be slightly changed. The primary simply informs backups about deleted components by piggybacking such information on regular messages. The backups then discard any information in regard to these components.

7.2 Recovery

Recovery is an important aspect of fault tolerance. In here, recovery means that a failed replica recovers or a new replica joins. It is important that recovery occurs online, i.e., while processing goes on in the rest of the system. When recovery takes place the recovered replica has to first receive the current state, and then will become a backup. Our solution is that one of the existing replicas, referred to as the *peer*, sends its current state to the joining replica. Either the current primary or any existing backup can serve as peer. For a backup, its current state is the current content in $COMP$ and RR. Hence, if choosing a backup as peer, we just need to send $COMP$ and RR to the recovered replica. For a primary, its current state includes the state on each component. It is not trivial to collect the state of a component during runtime. For a component, we have to collect its state when no execution is active on it. If two components are involved in the execution of the same client request, we have to collect both states after the execution is finished to guarantee consistency between them. Further requests to a component will be blocked until the collection is finished. Hence, choosing a primary as peer not only adds extra load to the primary but also might block normal processing. Its procedure is more complicated than if a backup is the peer. Whereas, choosing a backup as peer requires the system always has at least one backup.

Thus, it is preferable to choose a backup as the peer for simplicity and better performance. If there is more than one backup available, we need to choose one of them as peer. In our algorithm, the new replica first joins the FTG (fault tolerance group). All replicas receive the view change message and the new replica receives all messages delivered after the view change messages. Each backup who is willing to become the peer multicasts a *willing* message to all replicas using uniform-reliable delivery (to guarantee all or nothing) and total order delivery. The backup whose *willing* message is the first to be delivered will become the peer. When a backup receives the first *willing* message and it is the sender, it delays the processing of any new messages coming from the primary. It generates

a *recovery* message containing the content of $COMP$ and RR and sends it to the joining replica using point-to-point communication. While waiting for the *recovery* message, the joining replica might have already received messages from the primary (it starts receiving messages when the GCS delivers the view change). It enqueue them in a queue Q. Once the joining site receives the *recovery* message, it initializes its data structures accordingly. The *recovery* message might already contain the state of some of the messages in Q. Hence, these messages must be removed from Q before the backup algorithm can start processing messages from Q. In order to determine which messages to remove, we timestamp all messages.

In practice, there also exist conditions that require the primary to send the recovery message, e.g., only one replica keeps working while others crash. To adapt to these conditions, we propose a solution to allow the primary be the peer. Recall that we use uniform reliable delivery to multicast *committing* messages. This delivery mechanism requires that *committing* messages will also be delivered to the sender. Hence, we get the current state of all components by parsing *committing* messages on the primary. Although this solution requires the primary to use extra overhead to store and process *committing* messages, it avoids state collection at recovery time and does not block normal processing. In summary, if a replica joins and there is no other backup that could serve as peer, the primary sends a recovery message to the replica without sending a willing message.

7.3 Non-transactional Client Requests

In some applications, execution of a client request might not happen within the boundaries of a transaction. For example, in a J2EE environment, a method might have the transaction attribute `Supports`, `NotSupported`, or `Never`. This is mainly used for simple executions that never access the database, and hence, no transaction is required to be associated with the execution. Using the BMT scheme, the start/commit/abort commands have to be written into the method code. If this is not done, a client request calling the method is also not associated with a transaction. In another scenario, a transaction might only be started before the first database access and be terminated after the last database access but before the execution at the AS has finished. In the extreme case, the application does not set any transaction boundaries. In this case, the database executes each request

to the database within an individual transaction (when using JDBC, this is achieved by setting the autocommit flag to on). We can handle such a non-transactional client request r by assuming its action a is embedded within a pseudo transaction pt. pt is assumed to begin at the time the action a is started at the primary, and to commit when the action a completes but before the response to the client is returned. If r does not access the database, r's execution is transformed to the 1-1 pattern with pt being the only transaction. The state changes of the pseudo transaction are replicated immediately before returning the response. If the execution of r actually triggers one or more "real" transactions (which embed all database accesses), r's execution is transformed to the 1-N pattern, with pt being the outer transaction. Again, state changes of the pseudo transaction are replicated immediately before returning the response.

7.4 Accessing more than one Database

In our previous discussion, we assumed that an application only accesses one database. In practice, an application can access more than one database and then use the 2-phase commit protocol (2PC). In order to handle 2PC, we take an idea proposed in the e-Transaction system [46] that provides replication for stateless AS and adjust it to work with stateful AS. For that, we have to slightly change the commit handling of our algorithms (see Figure 5.2 (c)). The primary intercepts the first *prepare* request sent by the TM to a database and multicasts a *preparing* message to the backups using uniform reliable delivery before forwarding the request to the database. Then it intercepts the first decision (commit/abort) that the TM sends to one of the databases. In case of commit, it sends the *committing* message with uniform reliable delivery as in our previous algorithms before forwarding the commit to the database. After the transaction has terminated at all databases, the response is returned to the client and a corresponding *commit/abort* message is multicast to the backups (reliable delivery). No transaction id needs to be inserted into the database.

At the time the old primary crashes, the new primary might have received for a given transaction (1) not yet any message, (2) the *preparing* message, (3) the *committing* message, (4) the *abort/commit* message. In the first case, our failure assumptions guarantee an abort of the corresponding transaction at all databases. In case (2), some might have aborted the transaction, others

might be blocked in the prepared state. The new primary can now force all databases to abort the transaction if they have not yet done so. In case (3), some databases might have committed the transaction, others might be blocked, and the backup has received the component state changes. The new primary can now ask all databases to commit the transaction if they have not yet done so. In the last case, nothing needs to be done because all databases and the new primary have the correct state after transaction execution.

We can easily integrate the above solution into the 1-1/N-1/1-N algorithms. Since 2PC does not affect execution patterns, integration is not difficult. For each algorithm, we need to add the part to process *preparing* messages during normal processing. At failover time, instead of checking the marker, we have to consider the four phases described above.

7.5 Client and Database Crashes

So far, we have assumed that both clients and database are reliable. However, in practice, both of them might crash as well.

7.5.1 Database Crash

If a database crashes, the AS receives failure exceptions when it submits operations. It has now to wait until the database recovers. Upon recovery, the database aborts transactions that were active at the time of the crash.

From the perspective of the AS, this means that transactions for which it has not yet submitted the commit request, are aborted. Transactions for which the prepared request returned a failure exception might be aborted or in the prepared state. And transactions for which the commit request returned a failure exception might be aborted (if there was no 2PC), in the prepared state or committed.

The AS can determine the state of each transaction after recovery by looking for the $txid$ in the database or by asking the database whether a given transaction is in the prepared state. For an aborted transaction, the AS primary can easily replay the transaction in the 1-1 and 1-N patterns. In the N-1 case it has to forward the abort exception to the client replication algorithm with a request

to initiate the replay of the transaction. In the prepared case, the transaction can be terminated just as would be done during normal processing.

7.5.2 Client Crash

If a client crashes, a 1-1 or 1-N execution can simply finish the execution. A N-1 execution should abort the transaction if the client had not yet submitted the commit request because the AS server only has partial information about the transaction.

7.5.3 Replicated Database and Replicated Clients

Replicated Database Database replication has been widely used for fault-tolerance and performance. In many solutions, replications is mostly transparent to the application, i.e., the application is not aware of the fact that there are several database instances, each of them having a copy of the database. The degree of transparency and the level of consistency provided by the different solutions differ greatly. If the multi-tier architecture uses a replicated database layer, the AS layer must know the exact semantics provided by the database layer and be adjusted to work with the new semantics.

In [61] the authors take ADAPT-SIB at the AS layer, and a simplified version of the database replication solution proposed in [64] and show to what degree ADAPT-SIB has to be adjusted to work properly with the replicated database. In this case, the database replication solution is very powerful. It appears to the application nearly with the same semantics as a non-replicated database. The main difference is that a transaction might abort with a failure exception. This is the case when the database replica on which the transaction executes fails before the transaction commits. However, the AS primary, being the client of the database, is automatically connected to a new database replica. When the AS receives such an abort message due to failure, it can abort the corresponding transaction at the AS. Then, in case of the 1-1 or 1-N pattern, it can simply replay the transaction as is done in case of failover. In the N-1 pattern, it has to ask the client replication algorithm to initiate the replay.

Furthermore, in certain crash scenarios (e.g., both AS primary and the database replica the AS primary is connected to crash), inconsistencies could occur if certain failover operations at the AS

and the DB happen concurrently. In [108], we gave a demo of our ADAPT-SIB system working with the replicated database system presented in [64].

Replicated Clients In many cases, the client of the AS layer is actually a web-server (WS). In a well-designed system, the WS only calls the AS layer but not the database directly. The WS can also have state. The WS might start the transaction itself but it might also send simple requests to the AS, as we have discussed before. In order to provide fault-tolerance and load-balancing, also the WS-tier can be replicated [13]. The challenge now is to provide exactly-once execution across all three tiers: WS, AS and database. One problem is that request execution at the WS is often not embedded in a transaction, making it hard to reason about correctness.

Clara Huizink has looked into WS replication and its integration with ADAPT-SIB in her M.Sc. thesis [55].

Chapter 8

ADAPT-LB: Load Balancing Architecture based on ADAPT-SIB

The ADAPT-SIB replication tool only considers fault-tolerance. All replicas are members of a single *fault-tolerance group (FTG)*. There is one primary executing all requests, and all other replicas are backups. This addresses availability and reliability, but does not provide scalability compared to a single-node system. Since backup tasks typically require much less resources than executing the requests at the primary, the resources at the backups are wasted. In contrast, when replication is used for scalability, a load balancing algorithm uses server replicas as resources to execute different client requests. Ideally, the more replicas, the higher the maximum throughput the cluster can achieve. We can consider a group of replicas all executing requests as *load balancing group (LDG)*.

Considering that both fault-tolerance and scalability are important aspects of adaptability, this chapter proposes a unified architecture that provides both.

One major challenge of a unified replication architecture is to use replicas so that they serve both as resources for load-balancing and redundancy. That is, the question is how to build load distribution and fault-tolerance groups such that all resources in the system are exploited and enough redundancy is provided for fault-tolerance.

In this section, we present ADAPT-LB, a replication framework with both load-balancing and fault-tolerance modules. Each module relies on features of the other module to fulfill its tasks. The

main idea is that each replica is primary for the requests of some clients and is used as backup for other replicas. The solution has the following properties.

- **Load distribution** Each replica is member of a single large LDG and executes client requests.
- **Fault-tolerant execution** Each replica is primary of a small FTG and is backup in few other FTGs. As backup activity requires only few resources, the main capacity of each server is used for request execution.
- **Load-balancing** The system uses a truly distributed, lightweight load-distribution algorithm that takes advantage of the existence of FTG groups. It does not require the maintenance of load information and keeps communication overhead for load-balancing purposes low.
- **Dynamic reconfiguration** The system provides dynamic reconfiguration. When a replica joins the system, it joins the LDG and creates a new FTG for which it is primary. When a replica fails or is removed from the system, a backup replica takes over its tasks. As part of any join or leave operation, the FTG configuration is adjusted to guarantee that all FTGs have sufficient number of replicas and no replica is overburdened with backup tasks. Furthermore, the load-balancing module will quickly remove any load imbalance that might occur during reconfiguration.

As a summary, our unified solution distributes load across all replicas, handles failures transparently, and can easily grow and shrink with the demands, providing an ideal framework for self-provisioning.

8.1 Algorithm Overview

Assume the entire cluster consists of n ($0 < n$) replicas. In the ADAPT-LB architecture, the number of FTGs in the cluster is equal to the number of replicas in the cluster, because each replica is the primary in exactly one FTG, referred to as its *primary FTG*. Typically, it is enough for each FTG to have one or two backup replicas to tolerate the failure of a replica. Therefore, we let each replica join in one or two other FTGs as the backup. Each of these FTGs is called a *backup FTG* on the

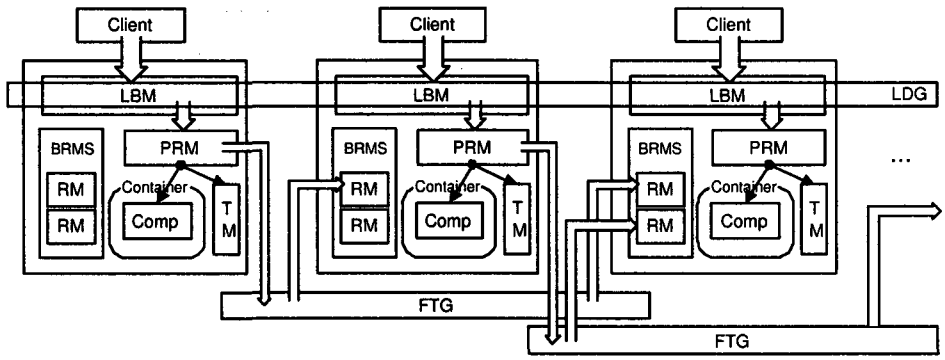


Figure 8.1: Unified architecture of ADAPT-LB

replica. Generally, each replica is backup in m ($0 < m < n$) FTGs, and hence each FTG has m backups. We refer to this as the m/m property. This property allows for a simple, yet powerful automatic reconfiguration mechanism, and also helps in load distribution. Figure 8.1 sketches the unified architecture of ADAPT-LB. Section 8.2 explains the components on the figure and describes the algorithm to initialize this setting.

Each replica processes client requests, using the ADAPT-SIB primary algorithm and sending replication messages (i.e., committing, committed, aborted messages) to the backups in its primary FTG. Additionally, it receives and processes replication messages from the primaries of its backup FTGs. The replica keeps the contexts of its FTG completely separated to avoid any interference.

When a client connects to the system, a session on one primary replica will be created, and all requests within this session will be handled by this replica. This guarantees that each request sees the current session state. Usually, sessions are randomly assigned to replicas. However, in case the current load of a replica is above a certain threshold, it can forward the request to its backups. If one backup is not overloaded it will accept the new client. Otherwise the request is forwarded to other FTGs recursively. Section 8.3 describes load-balancing in detail.

When a replica crashes, it leaves all its FTGs. For its primary FTG, the crash causes the failover process on backup replicas within the same FTG. According to the ADAPT-SIB protocol, one of the backups is chosen as the new primary of the FTG. However, since only one primary FTG is allowed on a replica and the chosen backup replica already has its own primary FTG, the chosen

replica has to merge the context of the backup FTG with the context of its own primary FTG. Then, the replica processes all incoming requests, no matter whether they are originally designated to the replica or they are resubmitted due to the crash, in the context of the primary FTG. Similarly, as the crashed replica also had backup FTGs, these have to be reconfigured. When a new replica joins or a failed replica recovers during runtime, it first initializes its primary FTG, and joins in other FTGs as backups. The primary FTG on the new replica automatically finds backup replicas in the cluster. Section 8.4 describes the algorithms to reconfigure the cluster when replicas fail or recover.

8.2 Cluster Initialization

Assume that the cluster starts up with a total of n ($n > 0$) replicas (note that the cluster size may change later dynamically). Each replica uses its address as a unique identifer. Each replica runs a *Load Balancing Manager* (LBM). When a replica starts up, its LBM first joins the LDG. Once all n replicas have joined, each LBM multicasts its replica identifier using total-order, uniform-reliable delivery. Each LBM receives the messages in the same order and stores the identifiers in an ordered list RL according to the delivery order. Each replica in the system is assigned an *order* number i, $1 \leq i \leq n$, which is the position of the replica in RL. We refer to the replica with order number i as r_i. Note that while the identifier of a replica does not change during its lifetime, the order number might change, as we see later. Once r_i has determined its order number i, it joins FTG_i as primary. If $i > m$, it furthermore joins FTG_{i-m} to FTG_{i-1} as backup. A replica with order $i \leq m$ joins FTG_{n-m+i} to FTG_n and FTG_1 to FTG_{i-1} as backup. For instance if $m = 2$, then r_3 joins FTG_1 and FTG_2, r_2 joins FTG_n and FTG_1, and r_1 joins FTG_{n-1} and FTG_n. Figure 8.2 depicts this circular setup of FTGs. Figure 8.1 shows the entire architecture. At each replica, PRM refers to the RM of the primary FTG, $BRMS$ refers to the array of RMs for the backup FTGs.

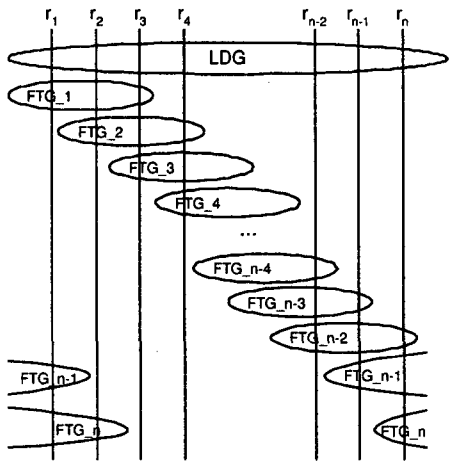


Figure 8.2: Initial setting with $m = 2$

8.3 Load Balancing Algorithm

After the initialization is completed, the workloads, namely client requests, are distributed on all replicas using the load balancing algorithm. While the algorithm uses generally a simply load distribution technique, it adjusts to variances whenever necessary.

8.3.1 Simple Load Distribution

Load balancing is performed when a client wants to create a new session. At this time, the client has to be assigned to a replica in the LDG. This replica becomes the primary replica for the client session, and all requests within this session are handled by this replica. This guarantees that each request sees the current session state. Load-balancing goes through two phases. The client has a pre-defined replica list CL (identifier list) that contains the replicas the client can potentially connect to. For correctness, only one replica on the list must actually be available. But a more accurate list has a positive impact on load distribution.

Our load distribution algorithm does not require extra message overhead. Instead, it is executed when the client creates the connection session with the AS, which is the standard procedure of the non-replicated AS to link a client with the AS (see Section 2.3.2). When the client wants to create a

connection session with the AS, it sends the request to any replica on CL. If it times out, it resends it to another replica until it succeeds. An available replica r_i receiving such a request becomes the load-balancer for this request. The LBM of r_i will decide on a replica r_j to create the session and serve the client by simply selecting a replica randomly from its RL list. The LBM of r_i returns the CRM code to the client. The message piggybacks an ordered list containing the identifiers of all replicas that are members of FTG_j (derived from RL) and indicates that r_j is the primary. The ordered list (called FL) on the CRM is used for fault tolerance, and is described in more detail in Section 8.4.3. In the second phase, when the client sends the first request, the newly installed CRM builds the physical connection session with r_j and r_j accepts the request within this session. Then, the CRM relays further client requests to r_j within the session. As seen in Fig. 8.1, the LBM of r_j intercepts all requests and dispatches them to the PRM. The message overhead for the session setup is the same as for standard J2EE involving one message round for the connection request (with r_i), and one for session creation (with r_j).

There exist many load-balancing algorithms that are more sophisticated than random, such as load or weight-based algorithms, or algorithms that take locality into account. These algorithms, however, require to exchange considerable information and maintain extra load information at each node. Most use a central load-balancer which we want to avoid. Additionally, random has shown to perform just as good as these load-balancing algorithms while having considerable less overhead. Since most enterprise clusters tend to over-provision to some degree, random will be good enough for most executions. Additionally, the random algorithm can be implemented locally at each replica. Furthermore, clients can be configured more easily since their CL do not need to be accurate. If CLs are stale, connection requests might not be equally distributed among replicas. However, the sessions themselves will always be distributed across all replicas since this is done at the server side.

8.3.2 Load Forwarding

Random replica assignment, however, does not work well if request execution times are not uniform since random assignment does not prevent that saturated replicas receive further clients leading to degradation of their performance.

We address this issue with a simple but effective *first-local-then-forward* (FLTF) mechanism.

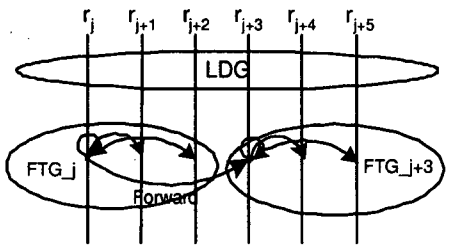


Figure 8.3: Forwarding a request

When the load of a replica is above a given threshold it will not accept any further client sessions but searches for another replica. Load could be measured as memory usage, CPU usage, response time, or the number of connected clients. We refer to a replica with a load below the threshold as a *valid* replica.

When the LBM of replica r_j receives a client session request and its load is below the threshold, i.e., r_j is a valid replica, it serves the client directly and establishes the session as described above. Otherwise, r_j multicasts a load query message lqm to all replicas in FTG_j in order to find a valid replica. When a backup replica in FTG_j receives the lqm message it checks its local load and sends an answer message back to r_j, which is positive if the load is below the threshold, otherwise it is negative. r_j chooses the r_k that was the first with a positive answer to serve the client. It returns to the CRM the list of replicas belonging to FTG_k. The CRM of the client refreshes its local FL, and sends a session request to r_k. In case of isolated overloads, contacting m other nodes will likely find a node that can accept new clients.

However, if there is no positive answer among the backups, r_j sends a *forward* message to the replica with the smallest order number larger than any order number in FTG_j, i.e., $r_{(j+m+1)\%n}$. $r_{(j+m+1)\%n}$ now repeats the process, sending a new lqm message in its own primary $FTG_{(j+m+1)\%n}$. Figure 8.3 shows this scenario of a forward. If a valid replica is found, $r_{(j+m+1)\%n}$ returns the information to r_j so that it can forward the relevant replica list to the client. If no replica is found, $r_{(j+m+1)\%n}$ could iterate the process by sending a *forward* message to $r_{(j+2m+2)\%n}$. We limit the number of iterations to a maximum T. If after T iterations no valid replica is found, a negative message is sent to the originator r_j which either accepts the client or refuses the connection. If T is

set 0, then no forward message is sent at all. Setting T low makes sense because if all nodes in r_j's neighborhood are saturated, then it is likely that the entire system is close to saturation and further forwarding will not help.

8.3.3 Discussion

A main benefit of our load distribution algorithm is that it is purely distributed without any central controller and can be easily implemented. It does not affect the fault-tolerance algorithm but takes advantage of the FTG infrastructure. Furthermore, load is checked in real time, and replicas can individually decide to take on further load or not. The main disadvantage is the overhead of the forward process.

One question is how often one has to forward to find a lightly loaded replica. The probability to find a valid replica within the m backup replicas of the local FTG is equal to the probability to find a valid replica within any m replicas in the cluster. If there are k valid replicas randomly distributed in the cluster, the probability p to find one of the k valid replicas within the m backup replicas is:

$$p = 1 - \binom{n-m-1}{k} / \binom{n-1}{k}. \tag{8.1}$$

Since each forward searches $m+1$ replicas (a new FTG), the probability p to find one of k valid replicas within the m backups of the initiator and T further forwards is:

$$p = 1 - \binom{n-m-1-T*(m+1)}{k} / \binom{n-1}{k}. \tag{8.2}$$

Assume the cluster has 100 replicas and $m = 2$. With 50 valid replicas and $T = 1$ we find a valid replica with more than 97% probability. With only 20 valid replicas, setting T to 3, the probability is still 92.8%. And even if there are only 10 available replicas, setting T to 5 will give us a probability of 86.3% to find a valid replica. Thus, this simple mechanisms provides a high success rate even for highly loaded clusters while keeping the message overhead low. Furthermore, there is no need to keep load or other information from other replicas.

Note, when all client requests trigger similar overhead and all replicas have similar resource

configuration, the random algorithm works fine. In this case, when one replica is overloaded, others are too, and forwarding only introduces unnecessary messages. But forwarding is useful if individual replicas are saturated for limited periods of time.

8.4 Reconfiguration

In Chapters 5 and 6, we discussed how the ADAPT-SIB algorithm performs failover and how new replicas can join an FTG as backups. Now we adjust these algorithms to fit with our overall architecture with the m/m property. When a replica fails or leaves, m existing FTGs now have only $m-1$ backups. Furthermore, when a new replica joins it needs m backups for its own primary FTG and it should join m existing FTGs as backup. We want to have a mechanism that automatically reconfigures the system back to an m/m configuration without any external intervention.

8.4.1 Server Crash

For simplicity of notation the following discussion assumes that a replica r_i fails where $i > m$ and $i + m < n$. When a replica r_i fails it leaves FTG_i, and a new primary has to be found for r_i's clients. Furthermore r_i is removed as a backup from FTG_{i-m} to FTG_{i-1}. Thus, these FTGs need new backups. Figure 8.4 illustrates the required changes. It shows the range of replicas around the failed replica r_i and the span of the FTGs. The bold lines are extensions, the dotted lines are removals.

The failover process at the server side is slightly different from the original ADAPT-SIB protocol. No new primary can be built for FTG_i, since r_{i+1} to r_{i+m} already have their own primary FTGs. Instead, the clients associated with FTG_i will be migrated to FTG_{i+1}, and FTG_i is removed. The main reconfiguration steps are as follows.

1. r_{i+1}, which is a backup in FTG_i, becomes the new primary for r_i's clients. It first blocks client requests. r_{i+1} then performs the failover on the components registered in FTG_i as described in the failover part of ADAPT-SIB. Then, it migrates these components into the context of FTG_{i+1}. Finally, r_{i+1} leaves FTG_i because this FTG ceases to exist. It now starts processing the blocked client requests.

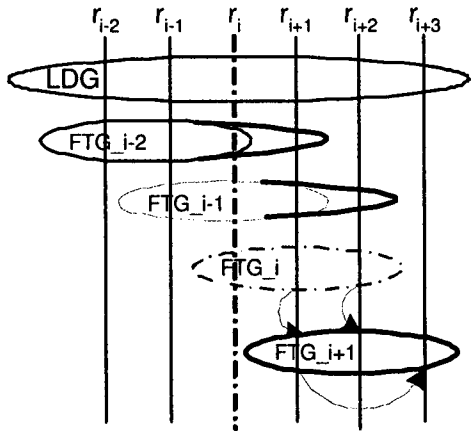


Figure 8.4: Crash scenario

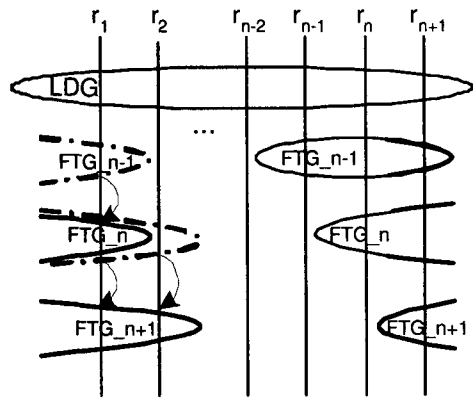


Figure 8.5: Recovery scenario

2. r_{i+2} to r_{i+m} are backups for FTG_i and FTG_{i+1}. They simply migrate the backup information they store for FTG_i to the context of FTG_{i+1} and leave FTG_i.

3. r_{i+m+1} is only backup of FTG_{i+1}. It has to receive the backup information for the clients that migrated from FTG_i to FTG_{i+1}. r_{i+1} sends this information to r_{i+m+1}.

4. Since replicas r_{i+1} to r_{i+m} have left FTG_i, they are now backups for only $m-1$ FTGs. Furthermore, FTG_{i-m} to FTG_{i-1} only have $m-1$ backups since r_i was removed from these groups. To resolve this, r_{i+1} joins FTG_{i-m} as backup, r_{i+2} joins FTG_{i-m+1}, etc. They use the normal recovery protocol of ADAPT-SIB.

Finally, each LBM removes r_i from its replica list RL, and decreases the order numbers of replicas r_{i+1} to r_n by one. Each replica can do this independently when it is informed by the view change protocol that r_i was removed from the LDG group.

8.4.2 Server Recovery

When a failed replica recovers or a new replica joins in a cluster of size n, it first joins the LDG, and all replicas are notified about this event. Each replica adds the new replica with order number $n+1$ to is replica list RL and considers it in its load-balancing task. r_{n+1} receives the replica list

RL from a peer replica. According to our setting, r_{n+1} must have a primary FTG and m backup FTGs. The reconfiguration changes are depicted in Figure 8.5.

1. r_{n+1} creates a new FTG_{n+1} and joins it as the first member.
2. r_{n+1} joins FTG_{n-m+1} to FTG_n as backups. It uses the recovery process of ADAPT-SIB as detailed in Section 7.2. These FTGs have now $m + 1$ backups.
3. Now, r_1 to r_m leave FTG_{n-m+1} to FTG_n respectively. The FTGs are now back to having m backups.
4. Finally, r_1 to r_m join the new FTG_{n+1}. They have again m backup FTGs and FTG_{n+1} has m backups. The recovery is fast, since this group is new and no backup information has to be transferred. The reconfiguration is complete and r_{n+1} starts accepting client requests.

8.4.3 Reconfiguration Effects on Client

Reconfigurations are completely transparent to the clients since the CRM takes care of reconnecting to a new replica if the primary it is connected to crashes. As mentioned before, when a client creates a session, the CRM receives an ordered list FL with the identifiers of replicas r_i, r_{i+1}, ... r_{i+m} with r_i being marked as the primary. For the CRM this is a simple list and it does not need to be aware that these servers constitute a FTG. If a backup leaves FTG_i or a new replica joins FTG_i, the client is not directly affected because the primary r_i is still available. Nevertheless, r_i piggybacks the new member list on the first response to the CRM after such a reconfiguration so keep the information at the CRM up-to-date. If the primary r_i crashes, the CRM chooses the next replica r_{i+1} on its FL to continue the session. When r_{i+1} receives the request, it processes it in its own primary FTG_{i+1} as discussed above. In the first response to the client, the new member list of FTG_{i+1} (i.e., r_{i+1}, r_{i+2},...) is returned to the client. Thus, the FL is always kept as accurate as possible.

Chapter 9

Implementation

This chapter describes our implementation of ADAPT-SIB and ADAPT-LB in an existing J2EE server. Our choice was on J2EE as it is more widely used than CORBA, and has more open source products than .NET. Our implementation has been integrated into JBoss AS [49], which is one of the most widely used open source J2EE products.

ADAPT-SIB and ADAPT-LB used the abstract concept of a component with volatile state, and assume that the replication tool can intercept requests. In order to prove the practicability of ADAPT-SIB and ADAPT-LB, the abstract components used so far have to be mapped to real components in a working system, and the replication tool has to be able to obtain control during the runtime of the system, in particular, before and after requests are executed. In the following, we first describe the J2EE architecture in more detail and then show how the replication tool can be plugged into the J2EE architecture. The implementation shows that the replication tool can be implemented with little changes at the client- and the database- tiers and without complex changes at the AS-tier.

9.1 J2EE Architecture

The general architecture of an AS and in particular of a J2EE based AS have been introduced in Chapter 2. This section describes three important parts of the J2EE architecture in more detail, namely, how a client sends requests to an EJB object, how the interceptor chain of the EJB container works, and how client requests are associated with transactions at runtime.

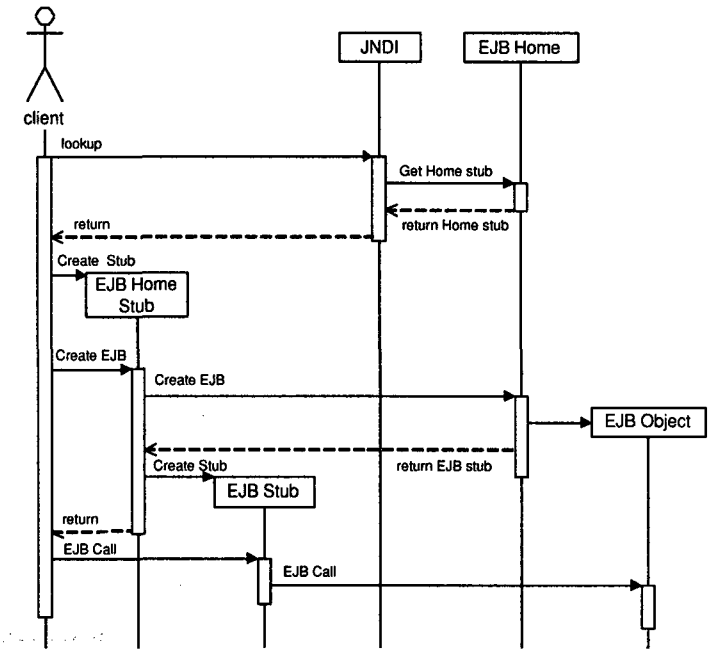


Figure 9.1: Lookup EJB from the client side

9.1.1 EJB Lookup

A client accesses EJB objects using the corresponding EJB stub. Figure 9.1 shows how a client gets the EJB stub. We have discussed this concept at a high level in Section 2.3.2 and then again in Section 8.3.1. Each EJB class has a corresponding home interface. It manages the life cycle of the EJB. When an EJB class is deployed on the AS, the home interface of the EJB class is registered with the name of the EJB class using the Java naming service (JNDI). When an outside client wants to access an EJB object, it connects to the JNDI service and looks up the EJB's name. If the name is found, the stub of the home interface for the EJB class is returned to the client via Java RMI (remote method invocation) [95]. The home interface provides remote methods to create a new EJB instance on the AS. A create method can return a remote stub of the EJB to the client side. The stub is the remote reference of the EJB object acting as the local proxy. Now, the client can send requests to the EJB object via the local EJB stub.

Although both session beans and entity beans can be accessed directly by the client using this

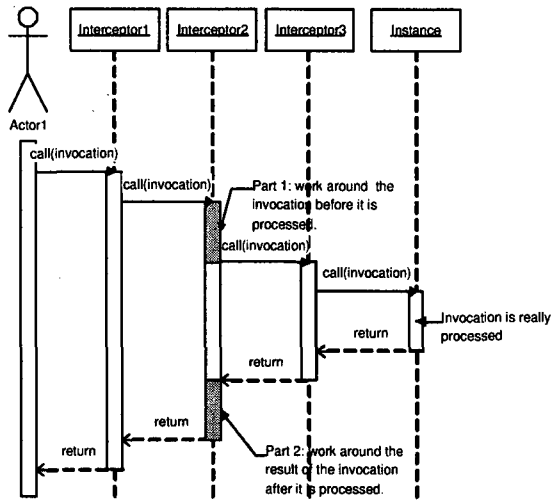


Figure 9.2: Interceptor chain

approach, the J2EE specification [94] suggests not to directly access entity beans from outside clients to guarantee correctness and better performance. Hence, we assume clients only access session beans. When a new session bean is created by a remote call from a client, a remote stub of the session bean is returned to the client, and thus a new session is created between the client and the AS. If the session bean is a SFSB, the state of this bean instance is bound to the session and is only accessible to the client. Our assumption that no concurrency issues occur on session state is based on this characteristic.

9.1.2 Interceptor Chain

As mentioned in Section 2.3.2, the EJB container consists of an interceptor chain, within which each interceptor is responsible for calling a certain service. Figure 9.2 shows how a request passes through the interceptor chain. A request passes through all interceptors in the chain before it reaches the destination EJB, and its response passes again through all such interceptors before it returns to the client. In JBoss, important interceptors in the interceptor chain of the EJB container are the *communication interceptor* that translates a request message sent from a client back into a method invocation to an EJB, the *component resolution interceptor* that finds the target EJB instance on

which the method will be invoked, and the *transaction interceptor* that associates the request execution with a transaction.

Taking advantage of the interceptor chain, we can easily manage to begin a service before the request is executed and to stop the service after the execution is finished but before it is returned. Considering the replication tool as a service, we can use a replication interceptor so that the replication tool can obtain control before and after a request is processed.

There could also be an interceptor chain at the client side. When downloading a remote stub of an EJB object, an interceptor chain is downloaded as well that can intercept client requests to the EJB object and the corresponding responses to the client side. The client side interceptor chain can help us to deploy the client part of the replication algorithms.

9.1.3 Associating Transactions with Requests

The transaction service is a service that is typically called on the EJB container via the transaction interceptor. The interceptor is responsible for associating a request with a transaction. Section 2.4.4 already introduced how the transaction interceptor associates requests with transactions for the CMT and BMT schemes. The transaction management is implemented in the transaction manager (TM), while the business logic is implemented in EJB objects. At runtime the TM and the EJB object must exchange information. For instance, the TM needs to know which database is required to be accessed by the business object, and the EJB method needs to know if a transaction has committed or not. In this case, the transaction interceptor is used to help the TM and the EJB objects exchange information.

If the client wants to determine the boundaries of the transaction, the client needs to download a remote stub of a user transaction object that represents the TM, and then explicitly start/commit/abort a transaction using the remote method invocation.

When a transaction is started, it first only executes at the AS. Only when the execution accesses the database, a DB transaction starts. That is, the TM starts the DB transaction with the first operation to the database and then terminates both at the AS and the database. If there is no database access, there is no DB transaction at all. Database access usually is controlled via a JDBC driver, which is provided by the database but runs at the AS. Hence, to obtain control over transactions, the

replication tool needs to intercept transaction commands to the TM (start/commit/abort) and JDBC commands sent to the database.

9.2 Implementation based on the Adapt Framework

Inspired by the interceptor mechanism, our partners at the Universitá di Bologna, Italy, with our help, implemented a pluggable module, called *ADAPT framework* [6] [1]. The ADAPT framework is an extension of a J2EE server, allowing replication algorithms to be plugged in. Towards the upper layer, the framework defines a set of APIs for the replication algorithm. The replication algorithm can be implemented using these APIs without considering the architecture of a certain AS. Below these interfaces, the framework for a specific J2EE AS implements a set of interceptors that gets control of the system during runtime without affecting the original functions of the server.

When an EJB is invoked at runtime, the framework transfers control to the replication algorithm implemented within the replication manager (RM). Through the APIs, the RM sees an abstract view of EJBs, invocations, and other elements of the server. The RM may perform any actions, such as setting component state or communicating with other replicas, before continuing the invocation. The main advantage of using the ADAPT framework, rather than modifying the server directly, is that it simplifies replication programming. The custom API isolates the replication algorithm from the details of the server implementation. Further, the algorithm is centralized in just a few classes, rather than scattered throughout the server system. Figure 9.3 shows how the framework separates the J2EE server from the replication algorithm implemented within the replication manager. Within the replication framework, RCS (remote component stub) and CH (component handle) represent abstract views of components on the client side and the server side. In general, the ADAPT framework has been designed to be used by various replication algorithms.

In order to implement ADAPT-SIB and ADAPT-LB, the replication framework should provide APIs to see EJBs and their states, to see invocations on EJBs, to see transactions, and to see database operations. In the following sub-sections, we introduce how this information is provided by the replication framework.

[1] Our major contributions are described in Section 9.2.4 and Section 9.2.5

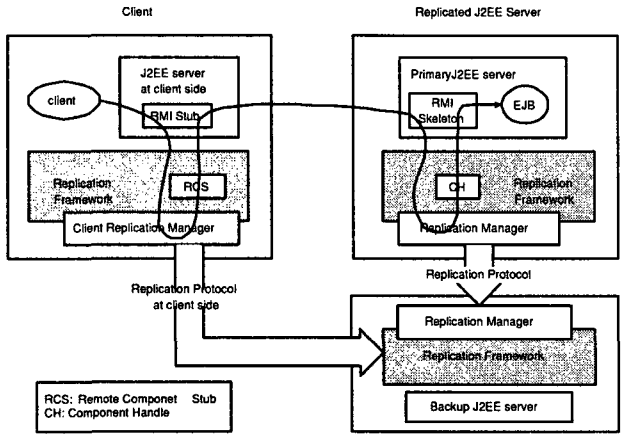


Figure 9.3: ADAPT framework separates replication algorithm from J2EE server

9.2.1 Components and States

The ADAPT framework API classifies components into three levels:

ComponentKind is the broadest classification. There are just a few kinds, fixed by the framework implementation: entity beans, stateful and stateless session beans.

ComponentType is a kind plus a name, specifying a particular "class" within the kind. The number of types depends on the applications deployed on the server.

ComponentHandle refers to a specific EJB instance. It consists of a ComponentType plus an instance identifier specific to that type. With an entity bean, the identifier is the primary key; with a stateful session bean, it is the session ID. The number of distinct handles depends on the number of EJBs invoked by the application.

All these classes may be transmitted between replicas. The classes also support comparison: two ComponentHandles, for example, test equal if and only if they refer to the same component instance.

An EJB instance is created by a call to one of the create methods of its home interface. The create method is intercepted by the ADAPT framework. After an EJB instance is created, the framework creates the ComponentHandle to represent the EJB instance. Then, in the framework, the information about the EJB instance can be received through the ComponentHandle.

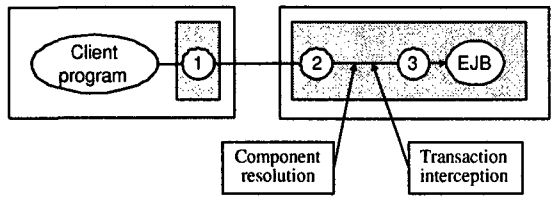


Figure 9.4: ADAPT framework intercepts the execution flow at three points

The ComponentHandle provides methods to test whether a EJB instance has state and to get the state. The state of an EJB instance is an opaque serializable object, which can be sent between replicas. To get the state of an SFSB, we use the passivation mechanism, saving the state to an array in memory instead of persistent storage. The state value is simply the serialized form of the object itself. The ComponentHandle can be replicated together with the array of the state value to backups. The replication of SFSB states is implemented in this way.

The ComponentHandle provides create methods that can re-create the EJB instance. It also provides a method to set the state value on the re-created EJB instance. Moreover, it provides the call method:

```
Response call(Request request).
```

This method can be used to eventually execute the intercepted request on the EJB instance represented by the ComponentHandle.

9.2.2 Invocation Interception

The ADAPT framework gets control of an execution by intercepting an EJB invocation at three points as shown in Figure 9.4. At each point, the replication algorithm may intervene, performing any computation or communication before or after continuing. The first point is within the client-side stubs. The interceptor at the first point is called the *client replication interceptor*. It is the entry point for the client-side replication algorithm, passing the control to the CRM (client replication manager). For example, when the interceptor intercepts a failure exception of the primary AS, it notifies the CRM to re-direct requests to the new primary. Both the second and the third points

are within the interceptor chain on the server side. The second point comes immediately after the invocation reaches the server, before the target EJB reference has been resolved. Intercepting here allows the replication algorithm to first re-create the EJB instance itself and set the state of the EJB instance, if necessary. This is used, e.g., by the Install-After-Failover strategy on the new primary. The interceptor at the second point is called the *early replication interceptor*. The third interceptor point comes just before the invocation passes to the target EJB instance; i.e., after the reference has been resolved, and all the EJB properties, such as security and transactions, have been set up. The interceptor at this point is called the *replication interceptor*. At this point, the control is passed to the RM (replication manager). Then, the RM can get the request and the corresponding response, get the EJB's state, get the current transaction, etc.

9.2.3 Requests and Responses

The replication interceptor transfers the control of a method invocation to the RM by triggering the invoke method of the RM as follows:

```
Response invoke(Request request, ComponentHandle component).
```

Within this method, the RM can get the state of the target EJB instance using the ComponentHandle parameter and get information about the request using the Request parameter. To invoke the target EJB instance, the RM calls the corresponding call method of the ComponentHandle described in Section 9.2.1. After the call method of the ComponentHandle returns the response, the RM can process the response (e.g., put it in the RR list), before the invoked method returns.

Generally, Request and Response are opaque to the replication algorithm. However, in a Request, the RM is allowed to read the name and the list of parameters of the method that is being invoked. This permits the replication algorithm to check whether two requests are identical. When the invocation completes normally, the Response encapsulates the return value. In this case, the replication algorithm cannot examine the content. However, when the invocation throws an exception, this is wrapped in a special Response which provides the details of the exception and identifies its source.

Application exception The exception was thrown by the component, i.e., by the application code. In this case, the replication algorithm should not examine the exception details, but should simply pass the Response back, where it will be handled by the calling component.

System exception Thrown by the system or framework, for example when the server crashes. The client-side replication can catch this and fail over to another server before returning to the caller.

Replication exception Thrown by the replication algorithm, presumably from some other point in the chain of invocation. In this case, the replication algorithm is free to examine the exception details and handle them as it chooses.

Both Request and Response can be tagged with headers. These are arbitrary key-value pairs, which are transmitted along with the content of the message. They are visible only to the replication algorithm. The key must be a string; the value may be of any class that can be serialized in the invocation. A common use for headers is to tag each request with a unique ID. This is to guarantee that each request will be executed exactly once, despite retransmissions and communication failures. For example, an ID can be set by the client-side stub in the client replication interceptor, before the request is sent the first time.

9.2.4 Transaction Interception

As mentioned in Section 9.1.2, there is a transaction interceptor in the interceptor chain to associate a request to a transaction. The transaction interceptor accesses the TM (transaction manager) of the AS using the JNDI service. It gets the reference to the TM by looking up "TransactionManager" in the naming service of the AS. In order to intercept transaction commands to the TM, the ADAPT framework provides the *TM wrapper* that wraps the TM by providing methods to intercept typical transaction commands, i.e., transaction begin, commit, and abort, as shown in the right side of Figure 9.5. Within these methods, the TM wrapper passes transaction commands to the TM by calling corresponding methods of the TM.

When the AS is started, the TM wrapper is started after the TM is started. Without changing the TM, the TM wrapper gets the reference to the TM by looking up "TransactionManager" in the

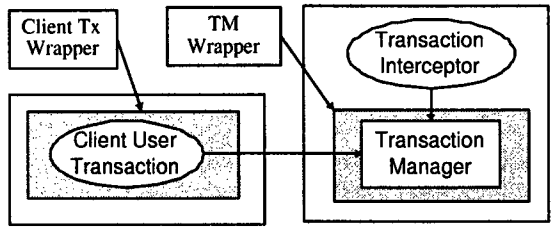


Figure 9.5: ADAPT framework wraps transaction manager and client-side user transaction

naming service. Then, the TM wrapper removes the binding of the TM and the name "TransactionManager" from the naming service. Instead, it binds itself with "TransactionManager" in the naming service. At runtime, when the transaction interceptor looks up "TransactionManager", it gets the reference to the TM wrapper but not the direct reference to the TM. This way, the TM wrapper intercepts all transaction commands that are sent from the outside, including the transaction interceptor. The TM wrapper has the reference to the RM, and hence can pass control of transactions to the replication algorithm before or after passing transaction commands to the TM. Thus, according to our replication algorithm, state propagation can be executed when a commit command is intercepted by the TM wrapper but before it is passed to the TM.

In the N-1 pattern, a transaction can be explicitly started, committed and aborted by the client. To do so, the client requires a remote stub referring to the TM. The remote stub is called *Client User Transaction*, which is binding to the naming service with the name "UserTransaction". The ADAPT framework provides the *Client Tx Wrapper* (CTW) to wrap the client user transaction, and binds it with "UserTransaction" in the naming service to replace the client user transaction. When a client is looking up "UserTransaction", the CTW containing the client user transaction is returned to the client as shown in the left side of Figure 9.5. Thus, the CTW can get control of transactions at the client side, and hence the client part of the N-1 algorithms gets control of transactions.

The wrapper approach, replacing original services in the naming service with wrappers is an easy way to plug our replication code into the system without modifying the original logic and code of the affected services.

9.2.5 JDBC Interception

According to our replication algorithm, a marker is inserted into the database before the commit to let the new primary check if a transaction eventually committed at the database or not. This insert only needs to be done if the transaction is an update transaction, i.e., database access involved in the transaction contains insert, update, or delete operations. In order to analyze database operations the ADAPT framework uses a JDBC wrapper to wrap the JDBC driver of the database. The JDBC wrapper implements the JDBC API defined in the JDBC 2.0 specification [97], and passes SQL statements to the real JDBC driver after analyzing them. The JDBC wrapper has the reference to the real JDBC driver by looking up the database source name in the naming service. Then, the wrapper replaces the entry of the JDBC driver in the naming service with itself. Thus, when EJB is accessing the database, it always gets the reference to the wrapper, and sends SQL statements to the wrapper. The wrapper checks each SQL statement whether it is an update operation, and if yes, marks the corresponding transaction as an update transaction. Then, the wrapper redirects each SQL statement to the real JDBC driver to trigger a database operation. Thus the replication algorithm can detect update transactions and insert a marker in the database accordingly.

9.2.6 Overall Architecture

Figure 9.6 shows the integration of the replication tool into JBoss using the ADAPT framework. The white boxes in the figure show the default JBoss components, the default client and the default database. The gray boxes show the building blocks provided by the ADAPT framework. The black boxes show the building blocks where the replication algorithm is implemented. Apparently, the client side replication algorithm is implemented within the CRM, and the server side replication algorithm is implemented within the RM.

The ADAPT framework gets the state of EJB instances and gets controls of requests and transactions in the standard J2EE environment. Using the ADAPT framework, the implementation of our replication tool just needs to focus on the implementation of the proposed algorithms. To plug the ADAPT framework into the JBoss application server, we have to implement the underlying APIs of the framework using APIs provided by JBoss. Two parts of the framework have to use APIs

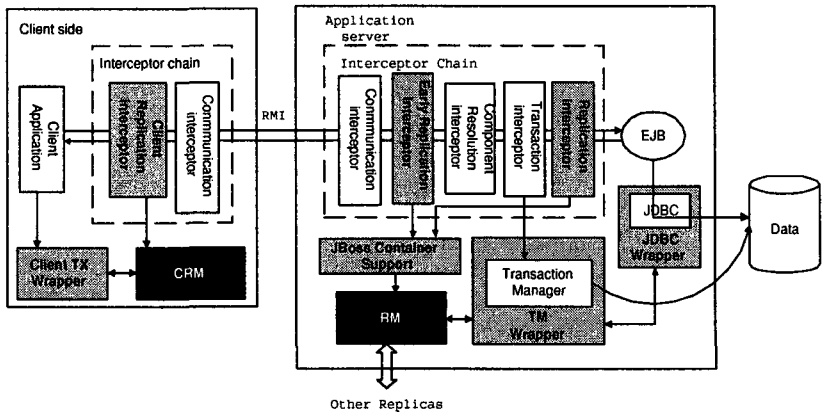


Figure 9.6: The implementation architecture of the ADAPT framework in JBoss

provided by JBoss. Firstly, all of the three interceptors, namely the client replication interceptor, the early replication interceptor, and the replication interceptor have to be extended from the abstract client interceptor and the abstract server interceptor of the JBoss implementation. Then, these interceptors are configured in the JBoss configuration file and are automatically loaded at runtime as other interceptors of JBoss. Secondly, the implementations related with the ComponentHandle, the Request and the Response have to use APIs provided by the JBoss container to get information about EJB instances and information about invocations to the JBoss container. All of these implementations concerned with the JBoss container are called *JBoss Container Support*. Our implementation of the JBoss container support is based on JBoss 3.2.3.

The transaction wrappers do not require the special support from the JBoss container since they implement the standard APIs defined in the Java transaction API specification [98]. The JDBC wrapper is based on the standard JDBC specification, and hence, does not require the special support from the JBoss container.

9.3 Implementation Issues

The client-side and server-side replication algorithms are implemented in the CRM and the RM respectively. The group communication system used for communication is based on Spread [1].

Our partners from Universitá di Trieste enhanced Spread and provided a set of Java interfaces, called JBora [12]. Hence, we use JBora for all inter-replica communication. In the ADAPT-SIB algorithm, each replica joins a single FTG group using the join method provided by JBora. In the ADAPT-LB algorithm, each replica joins a single LDG group, and also joins a set of FTGs depending on the configuration of the number of replicas required by a FTG.

The major issues relevant to the implementation are (i) how to extend the naming service for client applications to support a replicated AS, (ii) how to decide on the primary, and (iii) how the RM processes requests and transactions.

9.3.1 Extended Naming Service for the Replicated Application Server

As mentioned in Section 9.1.1, a client connects to the AS by looking up the home interface of an EJB. In order to do so, the client has a JNDI configuration with the destination address and port number of the AS. The standard JNDI service only supports the lookup on a single AS. In our implementation, we extend the JNDI lookup to support multiple AS replicas. On the client side, the JNDI configuration is now able to provide a list of AS replicas. When the client performs a lookup, the extended JNDI client randomly chooses a replica to send the lookup request to. If a failure exception occurs it chooses another replica from the list. Every AS replica, whether primary or only backup, has the same deployment of EJBs, and hence can generate the home interface of the target EJB and send the stub to the client side. At the same time, the CRM instance is generated on the server side. It contains the list of addresses of all available replicas with a flag on the address of the real primary that should be assigned to the client. Then, the CRM is downloaded to the client together with the stub of the home interface. When the client makes a request to the remote stub of the target EJB instance via the home interface, the CRM intercepts the request, and redirects the request to the real primary. Thus, the client communicates from now on with the real primary independently of who provided the EJB stub and the CRM object.

By taking advantage of the JNDI lookup, the ADAPT-SIB framework does not need any extra communication overhead to download the client part of our replication algorithm. The same holds for the ADAPT-LB framework. After the client submits the lookup request to an available AS replica, this replica decides randomly which AS replica will be the primary for the client, and

includes the relevant information with the CRM that is downloaded with the home stub to the client. Then, when the client creates the EJB object and calls EJB methods, the CRM automatically forwards these requests to the AS primary. This means, there is no extra communication overhead between a client and the server cluster.

9.3.2 Deciding on the Primary

The ADAPT-SIB algorithm has to decide on the primary AS when the system is started and after the current primary crashed. There are many ways to decide on a primary. We have not implemented anything special and assume all machines have the same power, i.e., there is no preference who is primary. At startup, a certain number of replicas is started, all joining the FTG group. Then, each multicasts a voting message to the FTG using total order. Due to the total order property, each replica receives the voting message in the same order. The replica, whose voting message is the first one received, is selected as primary. This way, every replica can make the same decision based on the voting messages. Lookup requests occurring during the select period are blocked until the primary is selected. After the primary is selected the lookup request return the CRM with the address of the primary. If other replicas join into the system after the primary is selected they will become backups. When a replica crashes, all other replicas are notified via the membership service of the GCS. If the crashed replica was the primary, the same voting mechanism is used as at system start-up.

The ADAPT-LB framework does not use this approach to decide the primary since each AS replica is a primary. At the beginning of the system, the LDG and FTGs are initialized using the algorithm described in Section 8.2.

9.3.3 Processing Requests and Transactions

On the server side, the replication algorithm is implemented in the RM as described in Chapter 5 and 6. The RM provides `begin`, `commit`, and `abort` methods to process transactions, whereby the control was obtained from the TM Wrapper. It also provides the `invoke` method to process requests, whereby the control was obtained from the replication interceptor. Moreover, it provides

the `enlistDBResources` method to record the database configured for a specific application.

In our algorithm description in Chapter 5, we use transaction ids to associate requests and their transactions in the replication algorithm. In the real implementation, we do not need to transfer transaction ids through transaction handling methods or request handling methods. Instead, we bind the transaction id with the thread that executes the transaction. Since requests have to be executed in the same thread as the transactions associated with the request, the RM can get the transaction id from the current thread. When intercepting a commit request of a transaction, the RM gets the state of all involved components using their componentHandles, and multicasts the state using the multicast API provided by JBora.

In order to insert the update marker into the database, the RM creates a special marker table in the recorded database when the system starts up. Then, when the replication algorithm needs to insert an update marker, the RM uses a standard SQL statement to insert the marker, which typically is the corresponding transaction id, into the marker table. After a crash and the selection of the new primary, failover executes simply as described in Section 5.2.

9.4 Summary

Our implementation of the replication tool does not depend on the JBoss implementation due to the use of the ADAPT framework. When changing to another J2EE AS, only the ADAPT framework has to be adjusted. Our implementation also does not require modifications to clients or database.

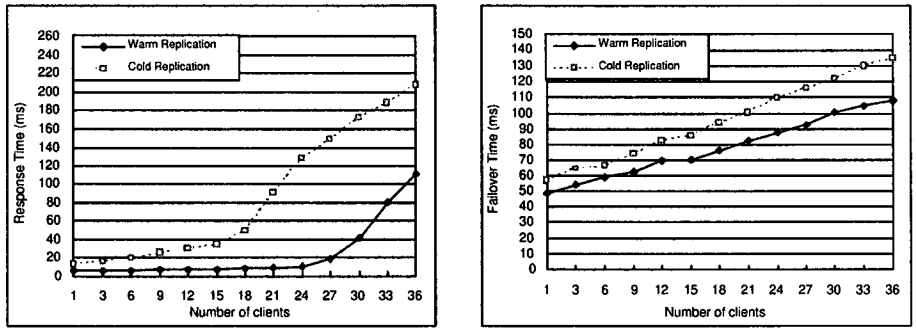
Chapter 10

Experiments and Evaluation

This chapter uses several suites of experiments to evaluate the performance of the ADAPT-SIB and ADAPT-LB frameworks. The evaluation of ADAPT-SIB focuses on the extra overhead caused by ADAPT-SIB. The evaluation of ADAPT-LB focuses on the scalability achievements. The replication algorithm and the load balancing algorithm provided by JBoss Cluster [60] are used as the reference system to evaluate performance of ADAPT-SIB and ADAPT-LB respectively. All replication algorithms are implemented based on JBoss 3.2.3. The backend database is DB2. As GCS, we use Spread [1] plus JBora [12].

10.1 Evaluation of ADAPT-SIB

This section uses four suite of experiments to evaluate ADAPT-SIB, our fault-tolerance framework. First, we use a micro benchmark to compare the performance of warm replication and cold replication. Then, we use the micro benchmark to show the impact of replication for different components and database access patterns. Then, we use the ECperf benchmark [93] to evaluate the performance of ADAPT-SIB replication tool on a more realistic application and compare it with JBoss's existing replication technique. The last experiment evaluates failover. We only use two AS replicas since the overhead at both the primary and the backup remains the same no matter whether there are two or more backups. Only the GCS might take longer for message delivery if there are more replicas. But the overhead between two, three, or four replicas is usually neglectable. All machines are 3.0



(a) Response Time (b) Failover Time

Figure 10.1: Performance comparison between warm and cold replication

GHz Pentium 4 with 1 GB of RAM running RedHat Linux. The configuration in all experiments consists of one machine emulating clients, one machine running the web server (if needed), two machines running JBoss application server 3.2.3 instances, and one machine running DB2 as our database system.

10.1.1 Performance Comparison between warm and cold Replication

In our first experiment suite, we use a simple test to compare the performance of warm replication and cold replication. Recall in warm replication, state changes are propagated to a running backup replica, while they are written to stable storage with cold replication, in our implementation to a DB2 database. We only consider SFSB, since we always use cold replication for EB. In the test, a client request triggers the execution of a single method of an SFSB within a transaction and the 1-1 algorithm applies. The main configuration variable is the number of clients. Each client is configured to submit 10 requests per second. However, since a client does not submit a new request before it receives the response for the previous request, if the execution time is longer than 100 ms, the real injection rate is smaller than 10/sec.

We compare the performance in regard to two aspects. First, we look at the average response time for each client request. Response time in this test includes the ordinary time to execute the request in the JBoss AS, the time to do state propagation at the end of each transaction (namely

each client request due to the 1-1 pattern), and the additional time to pass the ADAPT-framework. Hence, the response time implies the additional overhead induced by replication during normal processing. Figure 10.1 (a) shows the results. Response times increase slowly with increasing number of clients for both the warm replication and cold replication, and then increase sharply after saturation. As the time to execute the request is the same for both runs and both go through the same steps of the replication framework, the difference in the response times reflects the difference in costs between cold and warm replication. The main overhead of warm replication are serializing the state of the SFSB and propagating it to backups. The main overhead of cold replication are serializing the SFSB state and writing it into the database. The figure shows that warm replication has considerably better performance than cold replication. Before saturation, the response time with cold replication is typically double the response time with warm replication. At 15 clients, the response time with cold replication increases sharply due to CPU saturation, and the final saturation is after 24 clients. Warm replication reaches the saturation point later at around 30 clients.

The second performance aspect is failover time. In Section 7.1, we proposed three failover strategies. Cold replication can use the *Install-When-Failover* and the *Install-After-Failover* strategies. In here, we take the *Install-When-Failover* strategy since it shows better the costs of the failover steps as they all occur at once. We crash the primary after the system is running around 100 ms. Figure 10.1 (b) shows the corresponding failover time for both warm and cold replication when there was a certain number of clients connected to the old primary. The main overhead during failover for warm replication is to reconstruct all EJBs and restore the update-to-date state of each SFSB. Cold replication has to do the same but also requires additional time to read the logged state of each SFSB from the database. Both failover times increase slowly with increasing number of clients, because the number of SFSBs required to be reconstructed is increasing with the number of clients. The difference in failover times between cold and warm replication reflects the additional time to read the logged states of SFSBs from the database. This time gradually increases with the increasing number of clients, since as more clients exist more SFSBs are stored in the database. Please note that in this test the failover time is independent of the running time, because each client always accesses the same SFSB, and hence no new SFSBs will be accessed after all clients begin to submit requests. However, in a real application, a client might access more and

more EJBs during runtime. In this case, the failover time of the *Install-When-Failover* strategy will be affected by the running time. Our later experiments focusing on failover will show the effect. Although *Install-After-Failover* can shorten the failover time for both warm and cold replication, the additional overhead for cold replication to read from the database remains the same.

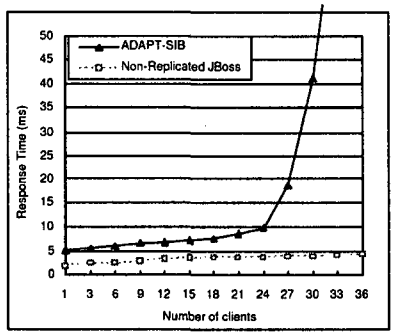
We conclude that warm replication has better performance than cold replication during both normal processing and failover. Hence, in the following experiments, we only focus on warm replication.

10.1.2 Component Analysis

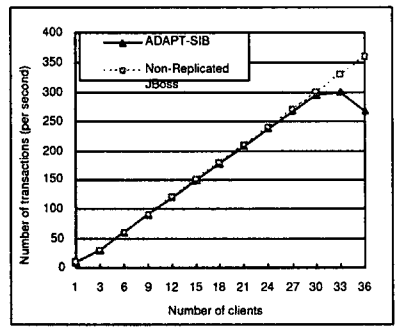
In our second experiment suite, we evaluate the overhead of replication for different components and component combinations. This experiment suite is implemented as 1-1 pattern. We evaluate the performance by comparing ADAPT-SIB with a non-replicated JBoss.

We consider the following cases. *Test 1:* No database access takes place. *Test 2:* Database access (update) takes place but no conflicts occur at the database. That is, different clients access different tuples. *Test 3:* Database access takes place and all transactions conflict. That is, all requests access the same tuple. In Test 1, a request triggers the execution of a single method of an SFSB. Test 2 and 3 have two different versions. In the first, a request executes only on one SFSB which makes the database call. In the second, a request calls a SFSB, which calls an EB to access the database. Each client submits 10 request per second, and the main configuration variable is the number of clients. Again, if the execution time is longer than 100 ms, the real injection rate is smaller than 10/sec.

Test 1: No database access Figure 10.2 shows (a) the average response time and (b) the throughput achievable with increasing number of clients. Response times increase slowly for both the replicated and non-replicated system. Below the saturation point, ADAPT-SIB (including the framework) has an overhead of around 4 ms. This is very low in total numbers, but means an overhead of around 100% for medium number of clients since response times are generally very small. This is the worst case scenario for our algorithm since it contains only SFSBs which all must be replicated. At 27 clients, response times increase sharply due to CPU saturation, and the final saturation is

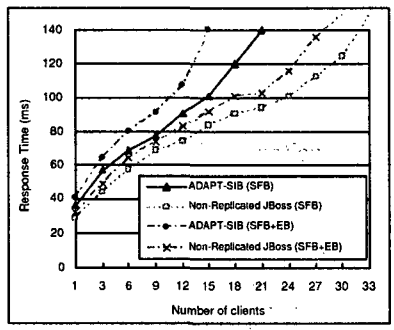


(a) Response Time

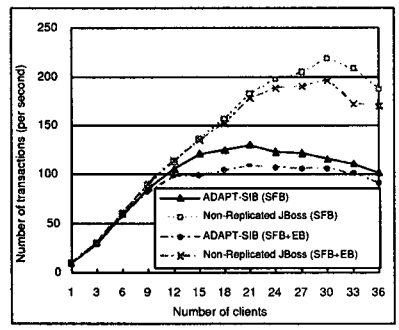


(b) Throughput

Figure 10.2: No database access

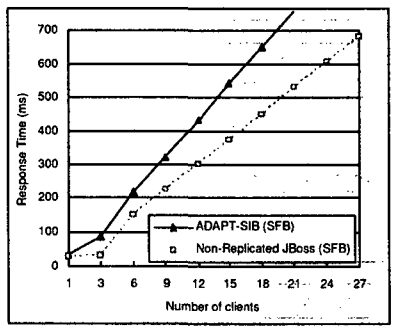


(a) Response Time

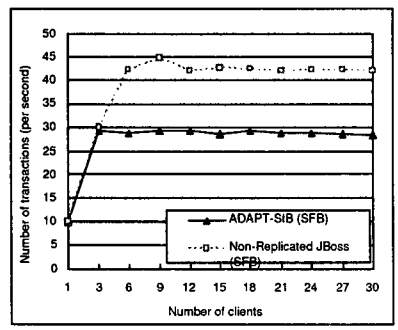


(b) Throughput

Figure 10.3: Conflict-free database access



(a) Response Time



(b) Throughput

Figure 10.4: Conflicting database access

after 33 clients. The non-replicated system does only saturate at around 66 clients again due to CPU overhead. Since the system is CPU bound, and the non-replicated system takes half the time to execute one request compared to the replicated system, it can execute double as many requests before saturation.

There are two solutions to improve the results of the replicated system. The first is to improve the implementation of the algorithm (e.g., data structures, access paths). This, however, can only succeed to a certain point. After that, alternative replication strategies have to be found, e.g., lazy replication.

Test 2: Conflict-free database access Figure 10.3 shows the results of test 2, in which transactions access the database but concurrent transactions never conflict. The figure contains graphs both for the SFSB only and SFSB/EB combinations. Let's first have a look at the SFSB only case. Compared to Figure 10.2 (a) for no database access, response times increase more steeply, and are generally higher. This is due to the database access. When the number of clients is smaller than 15, the overhead of ADAPT-SIB is stable at around 15 ms. The total time spent in ADAPT-SIB is higher than with no database access (4 ms) because the marker has to be inserted into the database (if a transaction does not update the database, no marker is inserted). In this scenario, 15 ms only mean an overhead of 20% for medium client numbers since transaction execution is generally long. With more than 15 clients, the time spent in the replication algorithm increases linearly with the number of clients and the throughput increases only very slowly 10.3 (b). At 15 clients, the CPU overhead is around 85%. After that, it does not increase fast because the system always waits for operations at the database to complete. The saturation point is at 22 clients. The non-replicated server reaches saturation with 33 clients.

When database access is filtered through EBs, response times both for the non-replicated and the replicated system are generally higher due to the EB overhead (see, e.g., [23], for a comparison of SFSB and EB). However, the relative performance is similar to the SFSB only case.

The conclusion is the easy observation that if the original system has high execution times, than the overhead of the replication algorithm has not such a big relative effect than with small execution times.

Test 3: Conflicting database access Figure 10.4 shows the results when all transactions conflict at the database. We only present the SFSB only case, since the effect of using EBs is similar to test 2. Generally, response times (Figure 10.4 (a)) are much larger than in test 2 due to the long blocking times at the database. They increase sharply with the number of clients for both replicated and non-replicated case. The difference between replicated and non-replicated system is bigger than in test 2 and also increases faster than in test 2. The reason is that ADAPT-SIB generally increases the execution time for each transaction. Assume transaction T1 holds a lock, and T2 and T3 wait for the lock. The time T1 needs longer to finish due to replication is also added to T2's and T3's execution time. Additionally, the longer execution time of T2 is added to T3's execution time. This means, waiting times are cumulative. We can also see that the maximum throughput (Figure 10.4 (b)) is only around 1/4 of the one in test 2 for both the replicated and non-replicated system due to the blocking.

As a conclusion, although the CPU is not saturated, the CPU overhead of replication limits its performance. Although the response time increase is due to longer waiting times at the database, it is caused by the computation overhead.

10.1.3 Evaluation of Different Execution Patterns

The previous experiments seemed to show that ADAPT-SIB had quite bad performance. However, the experiments were designed to show extreme cases, enabling us to understand the implications and influence of replication. In this section, we evaluate the performance of ADAPT-SIB on a more realistic application. We also pay attention to different execution patterns. ADAPT-SIB detects the execution pattern depending on the requests it intercepts, and automatically applies the corresponding algorithm.

To simulate a real application, we use the ECperf benchmark [93]. ECperf emulates businesses involved in manufacturing, supply chain and order/inventory management. The application is split into customer, manufacturing, supplier and corporate domains. The benchmark is quite database-heavy, i.e., the database is accessed frequently. The *transaction injection rate* (IR) is an indicator of the load submitted to the system (transactions per second). Results show the average response time of *order entry* transactions of the customer domain in milliseconds. Results are only measured over

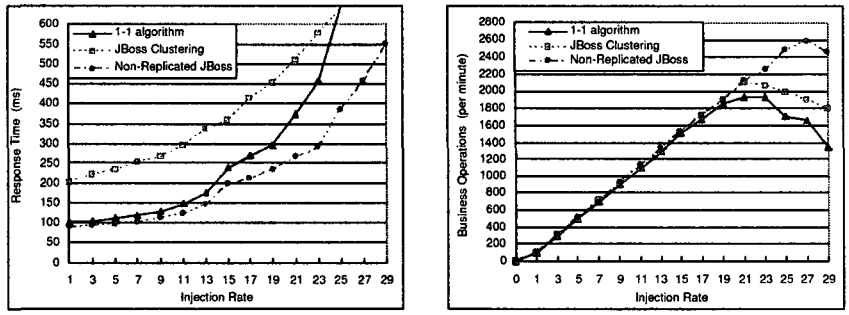
the steady state phase (10 minutes) of each test run.

Our evaluation compares (1) a regular, non-replicated JBoss server as baseline for comparison; (2) two JBoss server replicas using ADAPT-SIB; (3) two JBoss server replicas using JBoss's own replication solution called JBoss clustering. For both (2) and (3) one server was primary for all clients. JBoss has to basic configurations. In a fault-tolerance configuration, one machine in the cluster is primary and all others are backups. In this configuration, no load balancing property is provided. In a load-balancing configuration, all machines are able to process client requests. We discuss the load balancing configuration in more detail in Section 10.2. In this experiment, we configure JBoss clustering in the primary-backup model. As mentioned before in Section 3.1.1, JBoss clustering propagates state to backups on a component basis just before the component returns from a method call. Hence, if several components are called within one client request, several messages are sent. Moreover, please note that JBoss clustering cannot guarantee correctness even for the 1-1 pattern.

We look at the 1-1, N-1, and 1-N patterns individually to understand the impact of the particular mechanisms. The patterns are all used with accessing one database. At the end of the section, we have a test case with two databases that require 2PC.

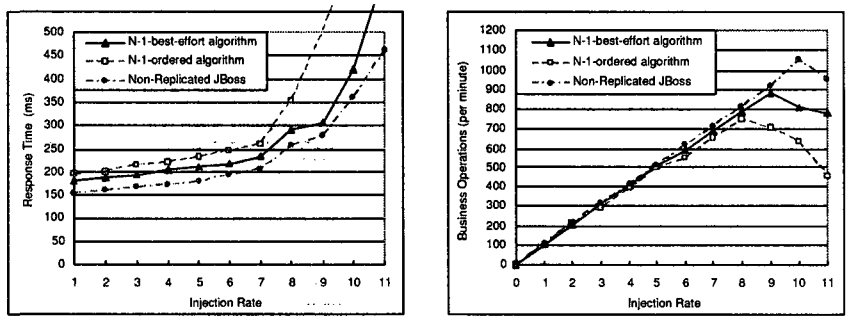
1-1 algorithm

In the original ECPerf benchmark implementation we used, all execution follows the 1-1 pattern. Figure 10.5 (a) shows the average response times of order entry transactions at increasing IR for the 1-1 execution pattern. The gap between the curve of ADAPT-SIB and the non-replicated system is the overhead of replication. At low load, the 1-1 algorithm adds 15 ms (15% overhead). As a comparison, [72] also indicates around 15% overhead for FT-CORBA (primary-backup) compared to non-replicated CORBA. JBoss clustering adds around 120 ms (120% overhead). The high overhead is due because it sends state after each method invocation while our approach sends one message per transaction. Response times for all setups increase steadily with increasing load until saturation points which is around 27 IR for the non-replicated JBoss, 23 for JBoss clustering and the 1-1 algorithm. More information about the saturation point can be found in Figure 10.5 (b) which shows maximum achievable throughput of the system with increasing IR. All systems saturate due to CPU overhead. Both the 1-1 algorithm and JBoss clustering saturate at 23 IR, while the



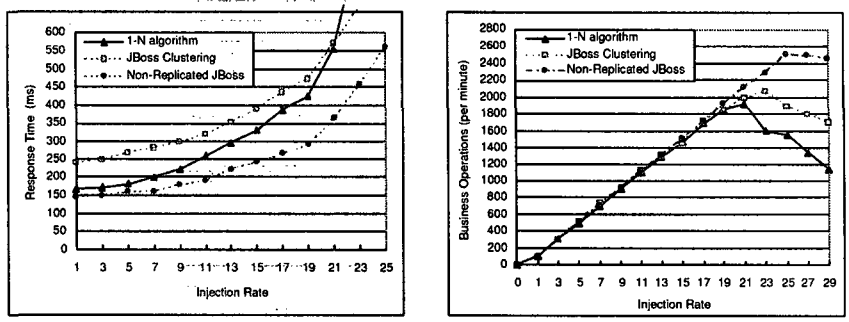
(a) Response Time (b) Throughput

Figure 10.5: ECperf comparison for the "1-1" Pattern



(a) Response Time (b) Throughput

Figure 10.6: ECperf comparison for the "N-1" Pattern



(a) Response Time (b) Throughput

Figure 10.7: ECperf comparison for the "1-N" Pattern

non-replicated JBoss saturates at 27 IR. As a summary, we believe the overhead of our approach is acceptable considering its strong correctness properties.

N-1 algorithm Figure 10.6 (a) shows the response times for the N-1 execution pattern. We modified the ECperf implementation so that each order entry transaction contains on average 5 order requests. The figure does not show results for JBoss clustering since response times are five times as high as in the 1-1 model. Response times are generally higher than for the 1-1 model shown in Figure 10.5 (a) since several client requests are included in one transaction. Compared to no replication, the N-1-best-effort algorithm adds again about 15% overhead while N-1-ordered adds 30%. The latter has higher overhead since it propagates the order in which database access takes place at the end of each client request. Considering that these are five additional messages, the overhead is quite small. This is true because the messages are small and only sent with reliable delivery. In regard to throughput shown in Figure 10.6 (b), all configurations saturate much earlier due to CPU overhead. N-1-ordered saturates at 8 IR, N-1-best-effort at 9 IR, and the non-replicated JBoss at 10 IR.

We would like to note that in ECperf many updates are on client related data with only few conflicts. Hence, even the N-1-best-effort algorithm provides exactly-once in most cases for this particular application.

1-N algorithm Figure 10.7 (a) shows the response times for the 1-N execution pattern. We changed the ECperf implementation such that each order entry request triggers an outer transaction which on average contains three inner transactions. Again, response times are generally higher than for the 1-1 execution pattern since now each order entry request includes several transactions. In absolute times, the 1-N algorithm takes more additional time than the 1-1 algorithm in Figure 10.5 (a) since we now have to send an additional uniform-reliable message for each inside transaction. In contrast, JBoss clustering adds the same time (120 ms) as in the 1-1 pattern since the replication mechanism is not related to transactions. In terms of throughput shown in Figure 10.7 (b), the 1-N algorithm saturates at 21 IR, JBoss clustering saturates at 23 IR, and the non-replicated JBoss saturates at 25 IR. The 1-N algorithms saturates earlier than JBoss because of the increased bookkeeping to guarantee all properties.

1-1 with 2PC Now, let's evaluate the extended algorithm which supports a transaction to access

Model	Algorithm	Response Time (ms)	Tx numbers (per second)
one database	Non-replicated JBoss	34.9	30
	ADAPT-SIB	40.3	26
more than one database	Non-replicated JBoss	103.5	10
	ADAPT-SIB	111.8	9

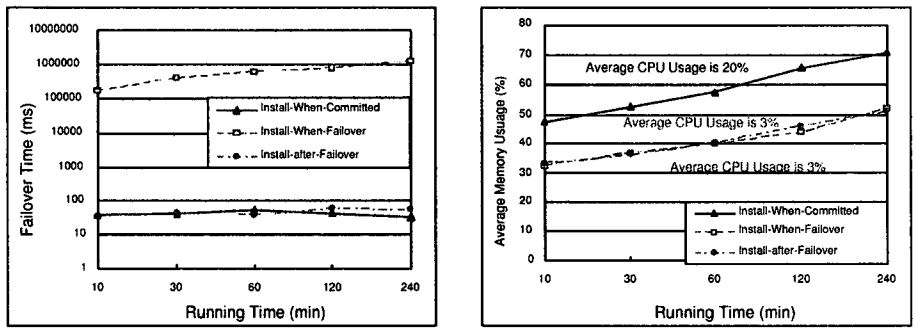
Table 10.1: 1-1 execution accessing one or more than one database

more than one database. Please recall that the extended algorithm is independent of the execution pattern. Hence, we can use the 1-1 algorithm as a sample pattern for the evaluation. For this experiment, we have not used the ECPerf but a simpler evaluation. A client submits one request to a SFSB which performs two database updates that either access the same database (no 2PC) or different databases (requiring a 2PC). Table 10.1 shows the average response time at a load of 10 transactions per second, and the maximum achievable throughput. Accessing one database, the ADAPT-SIB adds 5.4 ms to the response time of the non-replicated JBoss reflecting a 15% increase, while with a 2PC, ADAPT-SIB has an overhead of 8.3 ms (it has to send an additional *preparing* message) but this reflects an increase of only 8%. The maximum throughput for ADAPT-SIB compared to the non-replicated case is around 90% with a 2PC and 86% when one database is accessed. ADAPT-SIB performs, in relative terms, better with a 2PC than without because the total response times with a 2PC is so much higher than if no 2PC is necessary.

In summary, these experiments show that our solutions in general incur little overhead for all typical execution patterns on a realistic, i.e., quite diverse, workload. Our ADAPT-SIB replication tool clearly outperforms JBoss's clustering mechanism in all cases in terms of response time, and is similar in terms of saturation point.

10.1.4 Evaluation of Failover

In this section, we evaluate the overhead during failover, and compare the effects of different restore strategies. In this experiment, we only test warm replication as we have already shown in Section 10.1.1 that it has better performance than cold replication. We run ECPerf with an IR of 5 and crash the primary after different running times. Figure 10.8 (a) shows the time needed for



(a) Failover Time (b) Backup Overhead

Figure 10.8: Restore strategies comparison

failover (i.e., the time client requests are blocked and not executed). The failover time of *Install-When-Failover* is magnitudes longer than that needed by the other two strategies, and increases with the time the application server was running before the crash. When the primary was running for 10 minutes, failover takes around 2 minutes, when it was running 240 minutes, failover takes 20 minutes. This is because this strategy restores all components replicated at the time of failover, and the number of such components increases with the execution time. In JBoss, session beans are not automatically deleted when a user disconnects from the system. Instead the programmer needs to explicitly implement such methods, which was not done in the ECPerf implementation we have used. Hence, as time goes on, more and more such beans are in the system, are replicated, but are never removed. If beans were deleted at the primary, our algorithm would delete them at backups as well. In any case, the long failover time is clearly not acceptable since client requests that were submitted just before the crash or during the failover time are delayed during the entire failover period. For the other two strategies, failover time is always below 100 ms and is independent of the running time before the crash. This is because these two strategies do not restore components at the time of failover. For these two strategies, the main factor that impacts the failover time is the number of transactions for which the new primary received a committing message but no commit/abort message. For those transactions the new primary has to query the database at failover time. This number is independent of the running time and is usually very small because it reflects the number of transactions that were in the commit phase at the time of the crash. In regard to the

response time experienced by clients that submitted their request around the time of the crash, in the Install-Immediately strategy, these requests are delayed more or less by the failover time. Once failover has completed, client requests experience the same response time as before the crash. When using the Install-After-Failover strategy, response times for some requests are delayed beyond the plain failover time. These are the requests that access a component that needs to be restored. In our tests, such requests took around 1200 milliseconds compared to 110 milliseconds for consecutive requests on this component. This means, for each client connected to the system at the time of the crash, the first request to a component after failover takes long, but then execution is again fast.

In summary, although the discussion might simply be about an engineering problem, the large differences in performance show how important it is to consider an efficient implementation.

Figure 10.8 (b) shows the overhead on the backups during normal processing. The memory usage (simply measured using the UNIX top command) increases with the running time for all strategies as more and more components reside in the system (recall that beans are not removed when clients disconnect). *Install-Immediately* needs more memory than the others because it has all components created and installed. The other two strategies, in contrast, only store the serialized state information. The CPU overhead of *Install-Immediately* is around 20% CPU – needed to restore components, while the other two strategies only use on average 3% CPU to store the replicated information. With this, the backups can be used to do other work, as we do with ADAPT-SIB.

As a summary, we believe that *Install-After-Failover* is the best strategy. It has a much shorter failover time than *Install-When-Failover* and much smaller overhead at the backups during normal processing than *Install-Immediately*. Therefore, we use *Install-After-Failover* in the remaining experiments.

Figure 10.9 shows the failover time for the 1-1, 1-N, and N-1 algorithms after different running times of the ECPerf with an IR of 5. Additionally, the figure indicates the number of transactions for which the new primary needed to check in the database whether they committed. Since this number is small and independent of the running time, the failover time is always short. Comparing the failover times for the different algorithms, we can observe that 1-1 and 1-N have similar times, while failover in N-1 takes a bit longer. This is because the committing messages in the N-1 algorithm contain more information and hence need more time to be parsed during failover.

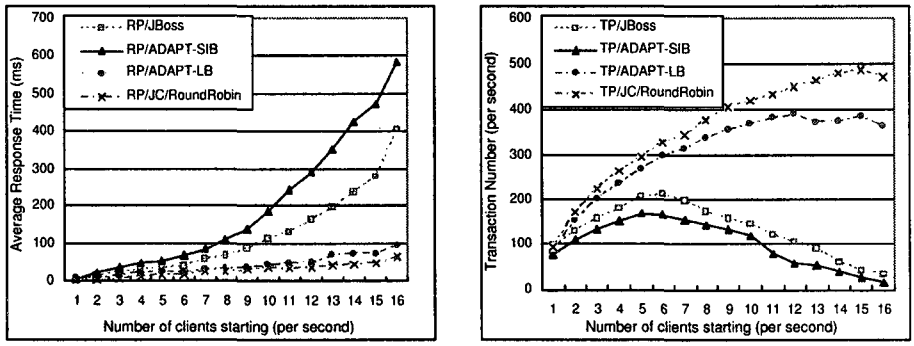
Algorithm	Running Time (minutes)	30	60	120	240
1-1	Number of committing transactions	2	1	3	2
	failover time (ms)	44	38	58	52
1-N	Number of committing transactions	1	2	1	2
	failover time (ms)	35	48	36	54
N-1	Number of committing transactions	2	3	1	4
	failover time (ms)	58	76	46	94

Figure 10.9: Failover time for different running time of ECPerf at 5 IR

Failover is impacted by the throughput at the primary. The more transactions are running at the same time, the more transactions might be committing at the time of the crash. We conducted a second experiment where we run ECPerf with increasing IR and crashed the primary after 30 minutes. The failover time increased from 20 ms at 1 IR to 380 ms at 21 IR for the 1-1 and 1-N algorithm and from 20 ms at 1 IR to 280 ms at 9 IR for the N-1 algorithm. Once the primary saturates, message propagation becomes bursty. As a result, just before the crash, the backup might have received many messages which must be processed first. In this case, failover time becomes much longer. Nevertheless, they remain short in absolute numbers.

10.2 Evaluation of ADAPT-LB

This section evaluates our unified framework ADAPT-LB, providing fault-tolerance and load-balancing. We use two benchmarks. We first use a micro benchmark to test the effects of ADAPT-LB on the AS, without considering the replication effects on the database. In the micro benchmark each client request performs operations on stateful session beans associated with the client but the database is not accessed. Clients connect to the system and then run for 10 seconds continuously submitting requests before they disconnect. All requests trigger transactions with similar load. We also use the ECPerf benchmark, which involves significant access to the database. Unless otherwise stated, experiments were performed on a cluster of 64-bit Xeon machines (3.0 GHz and 2G RAM) running RedHat Linux. In all our settings, each FTG consists of one primary and two backups. In this suite of experiments, we also use JBoss Clustering as the comparison framework. However, in this case, we use a load-balancing configuration. In our configuration, every machine in the cluster of JBoss



(a) Response Time (b) Throughput

Figure 10.10: Performance improvement

Clustering can process client requests, but no bean instances are replicated, i.e., fault-tolerance is not provided. JBoss clustering can be configured to integrate fault tolerance and load balancing together. Then, each machine will replicate its state to all other machines. But as we have seen in the previous section, the fault-tolerance mechanism of JBoss is very inefficient and does not provide correctness. Hence, we switch it off in this experiment.

10.2.1 Experiment 1: Basic Performance

In this experiment we have a first look at the performance of our unified architecture when no replicas leave or join the system. In Figure 10.10, `JBoss` refers to a standard single-node non-replicated JBoss application server without fault-tolerance. All other configurations use three machines. `ADAPT-SIB` refers to a system running the ADAPT-SIB algorithm but no load-distribution, i.e., there is one FTG but no LDG. `ADAPT-LB` refers to the unified architecture with one LDG using our load-balancing approach and several FTGs running ADAPT-SIB. `JC/RoundRobin` refers to a replicated cluster that uses the Round-Robin request distribution of the load balancing configuration of JBoss Clustering.

The figure shows response times in figure (a) and the throughput in figure (b) with increasing number of clients injected in the system per second. In the legend, throughput results are prefixed with `TP` and response times with `RP`. The non-replicated JBoss and ADAPT-SIB saturate when 6 clients are injected per second after which the throughput decreases. The maximum throughput

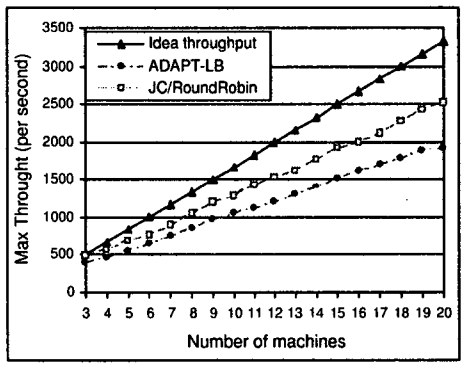


Figure 10.11: Scale-up homogenous setup

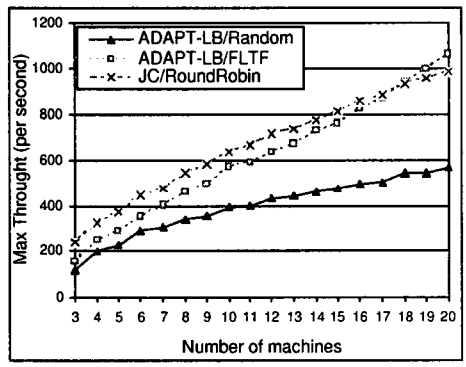


Figure 10.12: Scale-up heterogeneous hardware

for the non-replicated JBoss is 195 txn/sec and 130 txn/sec for ADAPT-SIB. As we have seen in the previous section, fault-tolerance adds overhead, and the maximum achievable throughput is smaller for ADAPT-SIB. The backups do not contribute to work distribution. The proposed ADAPT-LB is able to increase the throughput up to 12 clients with a peak of 380 txn/sec. JBoss' round-robing clustering can increase the throughput up to 15 clients with a peak of 480 txn/sec (`TP/JC/RoundRobin`). It outperforms ADAPT-LB since ADAPT-LB performs additional fault-tolerance measures.

Response times show similar results. For the non-replicated JBoss and ADAPT-SIB clients compete soon for resources. Response times increase early and deteriorate quickly after the saturation point. ADAPT-SIB has higher response times than a non-replicated JBoss, since the primary has to perform the state change collection and propagation. As we discussed before, ADAPT-SIB has similar response time behavior as other fault-tolerance algorithms [72]. In contrast, ADAPT-LB and JC/RoundRobin have low response times for all client numbers due to load distribution. Thus, each node is less loaded and can provide faster service. While ADAPT-LB has higher response times than JC/RoundRobin the difference is smaller than between ADAPT-SIB and the non-replicated JBoss, because ADAPT-LB is able to distribute the fault-tolerance overhead across all replicas. That is, our approach truly serves both fault-tolerance and scalability.

10.2.2 Experiment 2: Scalability

In this experiment we analyze whether our unified approach allows for sufficient scalability by running the micro benchmark on an increasing number of replicas. For each configuration we determine the maximum achievable throughput by adjusting the number of clients injected in the system per second. Figure 10.11 shows how the throughput increases when we increase the number of machines from 3 to 20. One graph shows the "ideal" throughput as the product of the number of machines and the maximum achievable throughput on a machine using the ADAPT-SIB primary algorithm (i.e, the machine is not backup at the same time). The two other graphs show our framework solution (ADAPT-LB) and JBoss' round-robin balancer (JC/RoundRobin). In both cases, throughput increases linearly with the number of replicas. Due to fault-tolerance activity on each node, ADAPT-LB achieves generally less throughput than JC/RoundRobin. But even JC/RoundRobin does not provide ideal throughput since the integration of load-balancing has its own overhead. In summary, our solution provides good scalability and at the same time provides fault-tolerance.

10.2.3 Experiment 3: Heterogeneity

Heterogeneity is a challenge for load-balancing techniques. We fist analyze the impact of heterogeneous hardware by replacing half of the machines with PIII machines (850 MHz and 256M RAM). As the forwarding mechanism described in Section 8.3.2 extends the simple random mechanism exactly for the purpose of handling heterogeneous environments, we analyze two different versions of the ADAPT-LB system. We use ADAPT-LB with only random load-balancing without forwarding, denoted as `ADAPT-LP/Random`, and ADAPT-LB with forwarding, denoted as `ADAPT-LB/FLTF`. We again compare with JBoss' round-robin load-balancer (JC/RoundRobin). Figure 10.12 shows the maximum achievable throughput when we increase machines from 3 to 20. In general, the throughput is lower than in the homogeneous environment (Fig. 10.11), since half of the machines are now weaker. ADAPT-LB/Random is the worst because random assignment ignores heterogeneity and fault-tolerance adds overhead. ADAPT-LB/FLTF and JC/RoundRobin have similar performance despite the fact that ADAPT-LB has the fault-tolerance overhead.

A more detailed throughput analysis helps to explain the results. Figure 10.13 shows the

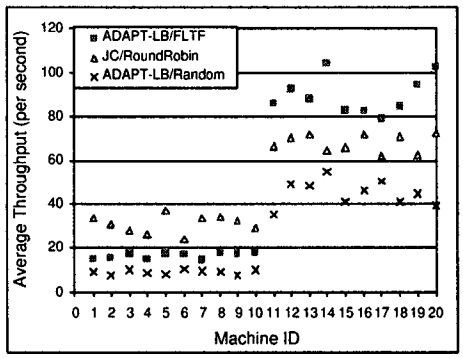


Figure 10.13: Throughput distribution

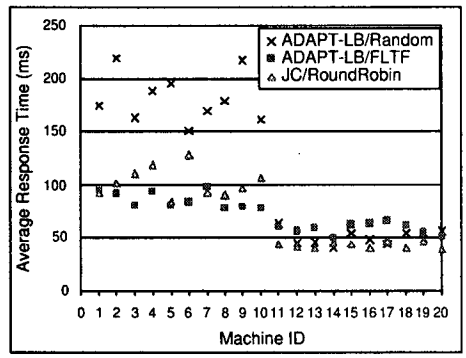
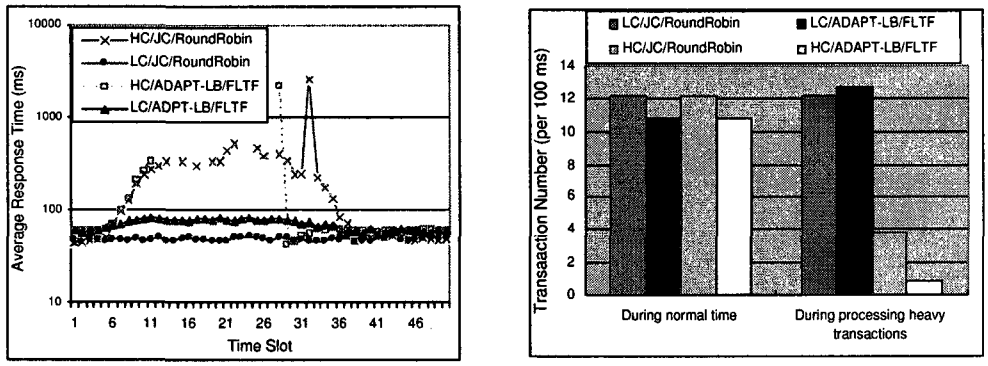


Figure 10.14: Response time distribution

throughput on each machine when the cluster contains 20 machines. Machines 1 to 10 represent the weak nodes. The figure shows that, compared to JC/RoundRobin, ADAPT-LB/FLFT has lower throughput on the weak and higher throughput on the strong nodes. This is because with ADAPT-LB/FLFT weak nodes forward requests that are then executed by the strong nodes. Thus, ADAPT-LB/FLTF compensates the overhead of fault-tolerance by a smarter load-balancing strategy which assigns more tasks to the stronger nodes.

Figure 10.14 shows the corresponding response time distribution. ADAPT-LB/Random has very high response times for weak nodes since they are saturated. Using JC/RoundRobin, weak nodes show worse response times than with ADAPT-LB/FLFT, which puts less load on the weak nodes. Strong nodes have low response times for all solutions because the bottleneck in the heterogeneous environment are the weak nodes. Since ADAPT-LB/FLFT puts more load on the strong nodes, it has slightly higher response times than the other two on these nodes.

In the second heterogeneity test, all machines are back to being the same but we add an additional very heavy client transaction to the micro benchmark with an average response time of around 2000 ms. We only compare ADAPT-LB/FLTF with JC/RoundRobin. At the beginning of this test, a cluster consisting of 6 machines runs the micro benchmark for about 30 seconds. Then we artificially inject the heavy transaction into the system. We refer to the machine executing the heavy transaction as HC. The other machines are denoted as LC. Figure 10.15 (a) has as x-axis time slots of 100 ms and as y-axis the average response times within a time slot. The heavy transaction starts at



(a) Average response time (b) Average throughput

Figure 10.15: Heterogeneous workloads

time slot 5. Before injecting the heavy transaction, HC and LC have the same response times which are higher for ADAPT-LB/FLTF because of the fault-tolerance overhead. Using JC/RoundRobin, the HC response times increase to around 400 ms after the injection of the heavy transaction because the HC machine becomes saturated. The response times on the LC group remain the same because they are not affected. Using ADAPT-LB/FLTF, response times on the HC machine increase for the first 5 time slots after the heavy transaction is injected. This represents transactions of clients that were already assigned to HC when the long transaction arrived. Then there is a long gap, since HC does not accept any further clients anymore according to the forwarding strategy. At time slot 27 the long transaction finishes (with a long response time). After that HC again accepts clients providing standard response times for them. While the heavy transaction is running on HC we observe longer response times at the LC machines because HC redirects clients to them, and thus they are more loaded. In total, response times are less affected using ADAPT-LB/FLTF compared to JC/RoundRobin which has unacceptable high response times for some of the clients.

Figure 10.15 (b) shows the throughput distribution during normal processing and while the heavy transaction is running. Without heavy transaction, ADAPT-LB and JC/RoundRobin have the same throughput on HC and LC machines. JC/RoundRobin has higher throughput because there is no fault-tolerance overhead. However, during processing the heavy transaction, the average throughput on the LC group of ADAPT-LB/FLTF is higher than the throughput of JC/RoundRobin, since

the FLTF algorithm forwards more transactions on the LC group. For the same reason, the throughput on the HC machine of ADAPT-LB/FLTF is less than the throughput of JC/RoundRobin on the HC machine. In absolute numbers, ADAPT-LB has 5*2 transactions / 100 ms more on LC and 3 transactions / 100 ms less on HC. Thus, in total, it has a higher throughput than JC/RoundRobin when the heavy transaction is injected.

In summary, our load-balancer can easily handle heterogeneous configurations and workloads being able to dynamically distribute the load according to the conditions on individual replicas even if inequalities only exist for short periods of time. It can achieve this without maintaining any global knowledge or knowing about the application semantics. Instead, it adds a simple forwarding mechanism exploiting the existing FTG groups.

10.2.4 Experiment 4: ECperf Benchmark

In this experiment, we conducted similar scalability experiments based on the ECperf benchmark in both the homogeneous environment and the heterogeneous environment (with two kinds of machines). In the ECperf benchmark, the throughput is measured by the business operations (BBops) processed per minute. Figure 10.16 shows the maximum achievable throughput for the ECperf benchmark for ADAPT-LB (here again only using the standard FLTF Strategy) and JC/RoundRobin. Generally, scalability is worse than with the micro benchmark and throughput even decreases with large number of machines. The reason is that ECperf contains considerable database access and the database becomes the bottleneck. One would need a stronger machine for the database server or database replication would be needed. In both homogenous and heterogeneous environments, JC/RoundRobin has slightly better throughput than ADAPT-LB with a small number of machines but behaves similar with many machines as ADAPT-LB can distribute the fault-tolerance overhead better with increasing number of machines. For heterogenous environments, the saturation point is later than in the homogeneous environment. The reason is that the throughput is generally lower, and thus, the database becomes the bottleneck later.

This experiment confirms our previous results. We are able to combine fault-tolerance with load

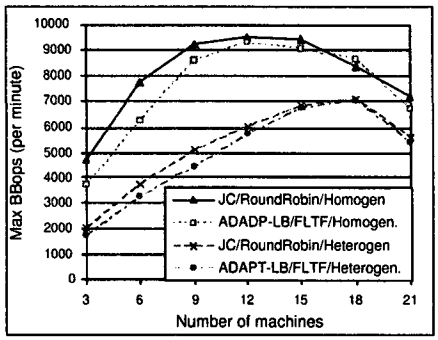


Figure 10.16: Scalability for ECperf benchmark

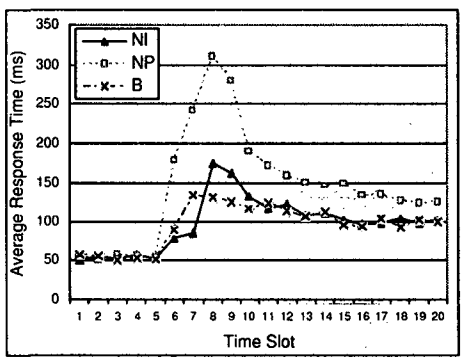


Figure 10.17: Reconfiguration: Failover

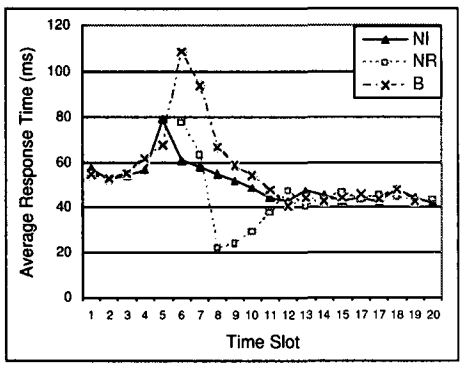


Figure 10.18: Reconfiguration: Recovery

distribution achieving scalability as long as the AS is the main bottleneck. The advantage of having the forwarding strategy compared to a simple round-robin strategy compensates the additional overhead for fault-tolerance.

10.2.5 Experiment 5: Reconfiguration

In this experiment, we show the detailed behavior of the ADAPT-LB system during and after reconfiguration. We first analyze the behavior of the system itself at the time of failover and recovery. Then, we compare the behavior of the ADAPT-LB system with the behavior of a typical alternative solution.

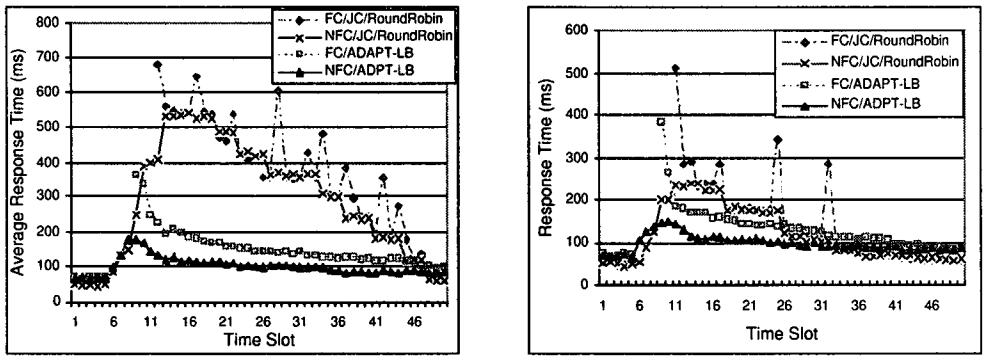
One goal of ADAPT-LB is to facilitate smooth and fast reconfigurations in order to be able to

provision the system dynamically if the need arises. The following experiments show the behavior of ADAPT-LB during and after reconfiguration. We use the micro-benchmark and a homogeneous environment.

We first look at node failures. We have a cluster of 6 replicas ($r_1, ..., r_6$) running the micro benchmark for about 30 seconds when replica r_3 crashes. We distinguish three types of replicas. NP (new primary) indicates the replica r_4 that takes over the clients of the failed replica r_3. B indicates replicas that have to reconfigure their backups (r_5 and r_6). NI indicates all other replicas on which the failure has no impact (r_1 and r_2). Figure 10.17 has as x-axis time slots of 100 ms, and as y-axis the average response time within a time slot. The crash occurs at time slot 5.

Before the crash, the average response time is similar in each group. After the crash, the response time on NP drastically increases because it requires considerable resources to perform the failover. This process takes about 300-400 ms. After that, the average response time is still higher than on the other groups because NP has now double the clients. The response time in B also increases (it shortly doubles) because the state transfer that takes place to include the new backups takes some of the resources. The recovery process to become a backup takes less than 100 ms. However, the response time on B remains higher and actually also increases on NI. The reason is that there is now one less replica in the system to execute requests. Furthermore, since NP is still higher loaded, the replicas in B and NI accept more of the newly injected clients. Eventually, once NP has stabilized, the system becomes balanced again. The average response time on all remaining replicas converges eventually to the same value (although not shown in the figure). However, this value is now higher because there is one less replica in the system to serve requests.

Figure 10.18 shows how the join of a new (recovered) replica affects the response time of the client transactions. At the beginning, the cluster has again 6 replicas. A new replica r_7 joins the system at the 5th slot (the time is counted when the server begins the LBM, and does not include the time to start the server from scratch). We distinguish between the new replica NR (i.e., r_7), replicas B (i.e., r_1 and r_2) that have to change $FTGs$ and replicas NI with no direct impact. For B we see that the response time doubles for a short period of time, similar to the failure case. The response time on the NI replicas is lightly affected due to sending recovery data to NR. NR starts accepting client requests at the next time slot after recovery. Shortly after recovery, the response time is not



(a) 6 machines cluster (b) 20 machines cluster

Figure 10.19: Comparison of failover operations

stable since some initialization process on the machine might not be completed yet. Eventually, all replicas converge to the same average response time. It is lower than before the join since there is now one more machine allowing for better load distribution.

As a final experiment, we compare ADAPT-LB with an alternative solution. Using JBoss' round-robin architecture without fault-tolerance, when a replica crashes, each client originally connected to the failed replica connects to any of the correct replicas and resubmits all requests from the beginning of the session. We call this solution the *re-execution solution.* Note that this only works correctly if the requests do not trigger changes on permanent components because these changes are already in the database and should not be applied again. Thus, the solution is not applicable for ECPerf but can be used for the micro-benchmark.

We again use 6 machines and crash one replica at time slot 5. This time, we group response times by client type. FC clients were originally connected to the crashed replica, and NFC are all other clients. Figure 10.19 shows the average response time over time. In ADAPT-LB, one replica takes all FC clients (and has additionally NFC clients). As long as this new primary performs failover, the FC are blocked. Therefore, there is a gap for FC clients where no response times are measured. Once execution resumes there is a peak in average response times as failover time is part of the response time. NFC clients on the new primary and also on other replicas are also affected, but much less (as discussed before). Response times for both FC and NFC quickly go back to normal levels. FC have still higher average response times than NFC clients since their primary

serves more clients in total. However, as new clients are prone to be distributed to other replicas due to the forwarding strategy, the response times of NFC and FC converge.

In the re-execution solution, FC clients are distributed over all replicas, which have to execute historical requests for them. This is a heavy task. For FC, the replay takes at least 10 time slots where no response is created. In general, response times stay high for *all* clients for a long time and only go down gradually because the machines are overloaded with the replay process. A peak in the graph of the FC clients occurs when one of these clients finishes the failover process, pushing the average response time for these clients up for this time slot. This shows that if replicas should be used for both load distribution and fault-tolerance then it is paramount to have a fast failover procedure as provided by ADAPT-SIB in order to keep the system responsive during failover times. A replay solution seems too expensive.

In Figure 10.19 (b), we repeat the same test for a cluster with 20 replicas. Two replicas are crashed at time slot 5 (namely the crash rate is about 10%). In this figure, the result of the proposed framework is similar to that of the previous test. However, the result for the re-execution solution is better than for the previous test. This is because the crash rate now is lower, and hence, each replica will be assigned with less FC clients.

This experiment has shown if replicas should be used for both fault tolerance and load distribution, then it is paramount to have a fast failover procedure in order to keep the system responsive during failure time. A replay solution will be expensive if it cannot be distributed well.

In summary, our approach can handle failures and recovery transparently and dynamically. Reconfiguration affects the client response times only shortly, and is relatively localized to few machines.

Chapter 11

Conclusions and Future Work

11.1 Summary

With application servers (AS) being a fundamental building block for web based applications, reliability, availability, and scalability are highly required guaranteeing 7/24 access and high performance. Replication is a common means to provide fault tolerance and facilitate load balancing. This dissertation presents novel AS replication solutions that are able to handle various execution patterns, that provide good performance, and that can be easily integrated into existing AS products.

11.1.1 Correct Replication for Different Execution Patterns

AS is a typical middleware that links clients and the backend database. As failure and AS replication do not only affect the AS tier itself but also the client tier and the database tier. Hence, it is necessary for an AS replication algorithm to consider correctness from the viewpoint of the entire system including the client tier, the AS tier, and the database tier. However, in practice, many AS replication solutions do not address this issue, and hence do not guarantee that the replicated AS behaves as a non-replicated non-faulty AS. We address this issue and identify a set of execution patterns that describe the behavior of an AS and its interaction between clients and the backend database. We observe that the crash of the AS affects clients because it interrupts the execution of client requests, while it affects the database because it interrupts the execution of transactions. Hence, we

define execution patterns in terms of different associations between client requests and transactions correlating clients, AS, and the backend database and thus, the entire system.

We formally describe the correctness requirements for AS replication based on different execution patterns, and accordingly propose a suite of replication algorithms in ADAPT-SIB framework. The general base of all algorithms in ADAPT-SIB is that one primary executes client requests and replicates state changes performed by a transaction to backups at commit time. We guarantee that consistency is maintained if the AS crashes at any time during execution. For most algorithms we provide a full proof of correctness.

11.1.2 Performance

Replication means extra overhead which is the price for fault tolerance. We carefully tune the performance of ADAPT-SIB algorithms and only add 15% extra overhead for JBoss AS, outperforming the JBoss Cluster replication algorithm. Furthermore, we provide a quick failover guaranteeing high availability.

In order to provide high scalability, we build ADAPT-LB load balancing solution based on ADAPT-SIB framework to offer an integrated solution providing both fault-tolerance and load balancing. In ADAPT-LB, each server acts as a primary to serve some of the client requests and at the same time stores state changes occurring on some other servers for fault-tolerance. It dynamically and transparently takes advantage of all resources in the cluster. It uses an effective, fully distributed load-balancing strategy that fits well with the completely distributed fault-tolerance solution. Our J2EE based implementation shows that we can in fact take advantage of the full power of all machines to provide fast response times, high throughput and high reliability, even in heterogenous environment .

11.1.3 Practicability

When developing our solution we always had a real system in mind. On the one hand, this led us to identify the advanced execution patterns. On the other hand this required to develop a solution that can also be implemented and integrated into a real system. Thus, our ADAPT-SIB replication

tool provides a suite of replication algorithms that can handle realistic execution patterns, without affecting the implementation on the client tier and the database tier. Furthermore, ADAPT-SIB tool is implemented as a pluggable module for the JBoss AS that does not affect the original implementation of the JBoss system and could be easily migrated to other AS systems.

Furthermore, ADAPT-LB solution provides a truly distributed load distribution algorithm. It is as simple and lightweight as content-blind approaches as it follows random distribution in underload situations but automatically switches to a content-aware approach when the load level of a replica reaches a critical threshold. Moreover, the content-aware mode takes advantage of the fault tolerance framework to distribute the load and does not introduce a complex or centralized distribution component. Hence, it can be easily implemented and deployed on an AS system that already uses ADAPT-SIB framework for fault tolerance.

11.2 Future Work

One major future work is to apply ADAPT-SIB and ADAPT-LB tool to other middleware systems. Nowadays, the multi-tire architecture is used everywhere. While the multi-tier architecture likely continues to dominate web-based applications, it is changing in structure, moving away from the traditional 4-tier architecture and consisting of multiple fine-grained or coarse-grained tiers. For example, on the micro level, a traditional AS might be deployed as a multi-tier distributed system, in which communication/message management, security management, service management and data management are deployed as different tiers. Each of these fine grained tiers manages a specific functionality, providing good resource distribution and easy maintenance. On the macro level, following the new trend of service composition based on service oriented architectures, new enterprise applications might be built upon composite multi-tier architectures, each tier of which is a normal 4-tier architecture. In both cases, we can see the trend that more and more middle tiers will be positioned in future applications and many of them are stateful. One of the most obvious flaws of this trend is that the more tiers a system has, the bigger the challenge to provide reliability, availability, and scalability. Hence, an interesting future work is to analyze how to apply ADAPT-SIB and the ADAPT-LB replication tools to these advanced middle-tiers architectures.

In different middle tiers, the content to be replicated and the right time point for state exchange might be different from the ADAPT-SIB. However, the general idea to take interaction between tiers into account and to analyze the correctness in terms of the effects on different tiers are still valid, considering that the request/response model is still widely used for many middle tier systems. Although there might not exist transactions, the non-transactional extension described in Section 7.3 could be applied to these systems and then be adapted according to the properties of their backend systems.

11.2.1 Enhancement to Handle Shared Data

In ADAPT-SIB replication tool, we don't consider the replication of shared data since it is assumed to be synchronized with the database. The correctness criteria developed in this thesis do not take the state changes on shared data into account. However, with this, caching cannot be exploited very well. The major relevant problem is how to handle concurrent accesses on the shared data. When all accesses are supposed to be executed on a single primary, replication of shared data can be easily handled. However, if accesses could occur on different replicas, e.g., in the load balancing approach, distributed concurrency control mechanisms have to be adopted while replicating shared data [84]. As well, the correctness criteria have to be adjusted to reflect the requirements of concurrency control.

11.2.2 Replication across a WAN

In ADAPT-SIB and ADAPT-LB tool, all AS replicas are assumed to be located within the same LAN. This makes sense for a typical 4-tier architecture since AS is the center server in such a system. However, when considering service composition, different middle tier systems of a composite architecture are normally distributed across a WAN. This might lead to replication across the WAN to achieve fast local access and high availability. However, coordination across a WAN is more expensive and using a GCS or having eager replication might not be feasible. Also, load-balancing might not be as important as providing clients services close to where they are. ADAPT-SIB and ADAPT-LB need to be revisited to see whether they can be adjusted for WAN purposes.

11.2.3 Extension of ADAPT-LB

The general load-balancing mechanism of ADAPT-LB can be applied to any middle tier which requires both fault tolerance and load balancing. If the replicated middle tier has a single backend database, this backend will quickly become the bottleneck while the middle-tier, using ADAPT-LB, can adjust to the load. In order to solve this problem, the backend has to apply a load balancing approach as well. There are two typical potential solutions. One solution is to use a replicated backend that has its own independent replication solution which is transparent to the middle-tier [64]. Another solution is to let each replica of the replicated middle tier have its own separate backend [84]. We are planning to investigate both solutions and their integration with ADAPT-LB.

Bibliography

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Center for Network and Distributed Systems, Johns Hopkins Univ., 2004.

[2] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. of Int. ACM/IFIP/USENIX Middleware Conf.*, pages 282–304, Rio de Janeiro, Brazil, 2003.

[3] C. Amza, A. L. Cox, and W. Zwaenepoel. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 230–241, 2005.

[4] M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. *SIGMETRICS Performance Evaluation Review*, pages 10–20, 2002.

[5] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.

[6] O. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, and H. Wu. A framework for prototyping J2EE replication algorithms. In *Proc. of Int. Symp. on Distributed Objects and Applications (DOA)*, pages 1413–1426, 2004.

[7] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software – Practice and Experience*, 33(8):1–31, 2003.

[8] R. Barga, S. Chen, and D. Lomet. Improving logging and recovery performance in Phoenix/App. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 486–497, 2004.

[9] R. Barga and D. Lome. Phoenix project: Fault tolerant applications. *ACM SIGMOD Record*, 31(2):94–100, 2002.

[10] R. Barga, D. Lomet, S. Paparizos, H. Y, and S. Chandresekaran. Persistent applications via automatic recovery. In *Proc. of the Int. Database Engineering and Applications Symp. (IDEAS)*, pages 258–267, 2003.

[11] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, pages 543–554, 2002.

[12] A. Bartoli, C. Calabrese, M. Prica, E. Antoniutti Di Muro, and A. Montresor. Adaptive message packing for group communication systems. In *Proc. of OTM Workshop on Reliable and Secure Middleware*, pages 912–925, 2003.

[13] A. Bartoli, M. Prica, and E. Antoniutti Di Muro. A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. Technical report, DEEI, University of Trieste, Italy, 2004.

[14] BEA Systems Inc. *BEA WebLogic server, release 7.0: Programming WebLogic enterprise JavaBeans*, 2002.

[15] T. Bennani, L. Blain, L. Courtes, J. C. Fabre, M.-O. Killijian, E. Marsden, and F. Taiani. Implementing simple replication protocols using CORBA portable interceptors and Java serialization. In *Proc. of Int. Conf. on Dependable Systems and Networks (DSN)*, pages 549–554, Florence, Italy, 2004.

[16] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596–615, 1984.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[18] P. A. Bernstein and E. Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 1997.

[19] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[20] K. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[21] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systemss*, 9(3):272–314, 1991.

[22] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems. second edition. ACM Press*, pages 199–216, New York, 1993.

[23] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proc. of ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 246–261, 2002.

[24] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial replication: Achieving scalability in redundant arrays of inexpensive databases. In *Proc. of Int. Conf. on Principles of Distributed Systems (OPODIS)*, pages 58–70, La Martinique, Frence, 2003.

[25] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 325–340, Montreal, Canada, 1991.

[26] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[27] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems concepts and design*. Addison Wesley, 2001.

[28] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[29] M. Cukier, J.. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Proc. of IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 245–253, 1998.

[30] X. Défago, P. Felber, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Proc. of Int. Working Conf. on Distributed Applications and Interoperable Systems*, pages 375–387, 1999.

[31] X. Défago and A. Schiper. Specification of replication techniques, semi-passive replication, and lazy consensus. Technical Report KS-RR-2002-001, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, February 2002.

[32] X. Défago and A. Schiper. Semi-passive replication and lazy consensus. *Journal of Parallel Distributed Computing*, 64(12):1380–1398, 2004.

[33] E. Dekel and G. Goft. ITRA: Inter-tier relationship architecture for end-to-end QoS. *The Journal of Supercomputing*, 28(1):43–70, 2004.

[34] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[35] A. Doudou, B. Garbinato, and R. Guerraoui. Tolerating arbitrary failures with state machine replication. In *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, 2005.

[36] S. Elnikety, S. G. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *Proc. of European Conf. on Computer Systems (EuroSys)*, 2007.

[37] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[38] Ensemble. Group communication systems, http://dsl.cs.technion.ac.il/projects/ensemble/.

[39] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA group communication service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[40] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA objects. *Lecture Notes in Computer Science*, 1752:254–276, 2000.

[41] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *Proc. of Int. Symp. on Distributed Objects and Applications (DOA)*, 2002.

[42] R. Friedman and E. Hadad. A group adaptor-based to CORBA fault-tolerance. In *IEEE distributed systems online, middleware*, 2001.

[43] S. Frølund and R. Guerraoui. A pragmatic implementation of E-transactions. In *Proc. of IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 186–195, 2000.

[44] S. Frølund and R. Guerraoui. X-ability: A theory of replication. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 229–237, 2000.

[45] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, 2001.

[46] S. Frølund and R. Guerraoui. E-transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering (TSE)*, 28(4):378–395, 2002.

[47] B. Garbinato, R. Guerraoui, and K. R. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering*, 2(1):14–27, 1995.

[48] D. K. Gifford. Weighted voting for replicated data. In *Proc. of the ACM Symp. on Operating Systems Principles (SIGOPS)*, pages 150–162, 1979.

[49] The JBoss Group. JBoss application server. http://www.jboss.org.

[50] Java Groups. homepage: http://www.jgroups.org/.

[51] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[52] V. Hadzilacos and S. Toueg. *Distributed systems*, chapter Fault-tolerant broadcasts and related problems, pages 97–145. Addison-Wesley, 1993.

[53] M. P. Herlihy and J. M. Wing. Axioms for concurrent objects. In *Proc. of ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987.

[54] K. S. Ho and H. V. Leong. An extended CORBA event service with support for load balancing and fault-tolerance. In *Proc. of Int. Symp. on Distributed Objects and Applications (DOA)*, pages 49–58, Antwerp, Belgium, 2000.

[55] C. Huizink. Replication across loosely-coupled tiers in a multi-tier architecture. Master's thesis, EPFL and McGill University, 2005.

[56] IONA Technologies PLC. *White paper Orbix E2A application load balancing and fault tolerance*, April 2002.

[57] IronFlare AB. *Orion application server clustering overview*, 2003. http://www.orionserver.com.

[58] ISIS. Group communication systems, http://www.cs.cornell.edu/Info/Projects/ISIS/.

[59] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, pages 227–238, 1987.

[60] The JBoss Group. *JBoss clustering*, 2002.

[61] B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Salas. Exactly-once interaction in a multi-tier architecture. In *(VLDB) Workshop on Design Implementation and Deployment of Database Replication*, 2005.

[62] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: A case study using Enterprise Java Beans. In *Proc. of IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 89–98, 2003.

[63] S. Kounev and A. P. Buchmann. Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In *Proc. of Int. Conf. on Very Large Data Bases (VLDB)*, pages 574–585, 2002.

[64] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *Proc. of Int. Conf. on Management of Data (SIGMOD)*, pages 419–430, Baltimore, ML, USA, 2005.

[65] M. C. Little and S. K. Shrivastava. Implementing high availability CORBA applications with java. In *Proc. of IEEE Workshop on Internet Applications*, pages 112–119, San Jose, California, July 1999.

[66] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal multivariate control for differentiated services on a shared hosting platform. In *Proc. of IEEE Conf. on Decision and Control (CDC)*, pages 3792–3799, 2007.

[67] X. Long, F. Yang, and S. Su. A CORBA design pattern to build load balancing and fault tolerant telecommunication software. In *Proc. of Int. Conf. on Communication Technology (ICCT)*, pages 1575–1579, Beijing, China, 2003.

[68] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high availability cluster communication. Technical Report CS94-14, 1994.

[69] C. Marchetti, R. Baldoni, S. T. Piergiovanni, and A. Virgillito. Fully distributed three-tier active software replication. *IEEE Transactions on Parallel Distributed System*, 17(7):633–645, 2006.

[70] MicroSoft Corporation. *Microsoft .NET framework 3.5*, November 2007.

[71] 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. *Communications of the ACM*, 39(4):54–63, 1996.

[72] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for CORBA. In *Proc. of Int. Symp. on Fault-Tolerant Computing*, pages 150–157, 1999.

[73] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *Proc. of Int. Conf. on Dependable Systems and Networks*, pages 261–270, 2001.

[74] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering*, 17(2):103–114, 2002.

[75] Object Management Group. *Fault tolerant CORBA specification*, December 1999.

[76] Object Management Group. *Transaction service specification*, September 2002.

[77] Object Management Group. *CORBA component model specification*, April 2006.

[78] Oracle Corporation. *Oracle9i applicatoin server*, release 2 edition, April 2002.

[79] O. Othman, C. O'Ryan, and D. C. Schmidt. Strategies for CORBA middleware-based load balancing. In *IEEE Distributed Systems Online*, 2001. http://www.computer.org/dsonline.

[80] O. Othman and D. C. Schmidt. Optimizing distributed system performance via adaptive middleware load balancing. In *Proc. of ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM)*, Snowbird, Utah, USA, 2001.

[81] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[82] J. F. Paris and D. E. Long. Efficient dynamic voting algorithms. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 268–275, 1988.

[83] M. Pasin, M. Riveill, and T. S. Weber. High-available enterprise JavaBeans using group communication system support. In *Proc. of the European Research Seminar on Advances in Distributed Systems ERSADS*, Berliner(Frolic),Italy, May 2001.

[84] F. Perez-Sorrosal, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme. Consistent and scalable cache replication for multi-tier J2EE applications. In *Proc. of Int. ACM/IFIP/USENIX Middleware Conf.*, pages 328–347, Newport Beach, CA, USA, 2007.

[85] Pramati Technologies Private Limited. *Pramati server 3.0 administration guide*, 2002. http://www.pramati.com.

[86] L. L. Pullum. *Software fault tolerance: Techniques and implementation.* Artech House, Norwood, MA, 2001.

[87] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294– 324, 1998.

[88] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[89] F. B. Schneider. Replication management using the state-machine approach. In *Distributed Systems. second edition. ACM Press*, pages 169–197, New York, 1993.

[90] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3):325–363, 1995.

[91] A.V. Singh, L.E. Moser, and P.M. Melliar-Smith. Integrating fault tolerance and load balancing in distributed systems based on CORBA. In *Proc. of European Dependable Computing Conf. (EDCC)*, 2005.

[92] J. G. Slember and P. Narasimhan. Living with nondeterminism in replicated middleware applications. In *Proc. of Int. ACM/IFIP/USENIX Middleware Conf.*, pages 81–100, 2006.

[93] SUN Microsystems Inc. *ECperf^{TM} specification, Version 1.1.*

[94] SUN Microsystems Inc. *JAVA 2 platform enterprise edition specification, Version 1.3.*

[95] SUN Microsystems Inc. *Java remote method invocation.*

[96] SUN Microsystems Inc. *Sun Java^{TM} system application server enterprise edition 7.*

[97] SUN Microsystems Inc. *JDBC 2.0 standard extension API*, December 1998.

[98] SUN Microsystems Inc. *Java transaction API specificatoin, Version 1.0.1*, April 1999.

[99] SUN Microsystems Inc. *Java transaction service specification, Version 1.0*, December 1999.

[100] SUN Microsystems Inc. *Enterprise JavaBeans^{TM} specification, Version 2.0*, October 2000.

[101] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[102] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[103] H. Wang and M. Bransford. *Server clusters For high availability in WebSphere application server network deployment edition 5.0.* Software Group, IBM Corporation, release 5.0 edition, April 2003.

[104] M. Wiesmann, F. Pedon, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in database and distributed systems. In *Proc. of the Int. Conf. on Distributed Computing Systems ICDCS*, pages 464–474, Taipei, Taiwan, R.O.C., April 2000.

[105] Inc. WinterGreen Research. Application server market opportunities, strategies, and forecasts, 2004 to 2009. Technical report, 2004.

[106] H. Wu and B. Kemme. Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In *Proc. of IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 95–105, Orlando, FL, USA, 2005.

[107] H. Wu and B. Kemme. Showing correctness of a replication algorithm in a component based system. In *Proc. of Int. Database Engineering and Application Symp. (IDEAS)*, 2008.

[108] H. Wu, B. Kemme, A. Bartoli, and S. Patarin. A replication toolkit for j2ee application servers. In *Software Demonstration at the Int. ACM/IFIP/USENIX Middleware Conf.*, 2005.

[109] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Proc. of Int. Symp. on Distributed Objects and Applications (DOA)*, pages 1376–1394, 2004.

[110] M. Wu and X.-H. Sun. The GHS grid scheduling system: Implementation and performance comparison. In *Proc. of Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 25–29, 2006.

[111] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Transactions on Parallel Distributed System*, 16(3):219–233, 2005.

[112] W. Zhao. A lightweight fault tolerance framework for web services. In *Proc. of the IEEE/WIC/ACM Int. Conf. on Web Intelligence*, pages 542–548, 2006.

[113] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Design and implementation of a pluggable fault tolerant CORBA infrastructure. In *Proc. of Int. Parallel and Distributed Processing Symp.*, pages 343–352, Fort Lauderdale, California, 2002.

[114] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proc. of Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 290–297, 2002.

[115] W. Zhao, L.E. Moser, and P.M. Melliar-Smith. Unification of transactions and replication in three-tier architectures based on CORBA. *IEEE Transactions on Dependable and Secure Computing*, 2(1):20–33, 2005.