Universidad Politécnica de Madrid

Facultad de Informática

Middleware for High-Available and Scalable Multi-Tier and Service-Oriented Architectures

Francisco Pérez-Sorrosal
M.Sc. in Computer Science

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

2009

Departamento de Lenguajes y Sistemas Informáticos
e Ingeniería de Software

Facultad de Informática

Middleware for High-Available and Scalable Multi-Tier and Service-Oriented Architectures

Francisco Pérez-Sorrosal

Advisor: Marta Patiño-Martínez
Doctor of Philosophy in Computer Science

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

2009

Tribunal nombrado por el Mgnfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día ..... de ..... de 200...

Presidente: _____

Secretario: _____

Vocal: _____

Vocal: _____

Vocal: _____

Suplente: _____

Suplente: _____

Realizado el acto de defensa y lectura de la Tesis el día ..... de ..... de 200... en la E.T.S.I. /Facultad.....

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

Treat a man as he appears to be, and you make him worse. But treat a man as if he were what he potentially could be, and you make him what he should be.

– Johann Wolfgang von Goethe

To everybody that wants to take it personally.

Acknowledgements

Firstly, I am grateful to my parents, family and friends for their support and encouragement during all this time.

Then, I would like to acknowledge my present and past colleagues at LSD –Damián Serrano, Rodrigo Dias, Marcio Shoiti, Jorge Salas, Jakša Vučković, Alberto Paz, Andrea Ricci, José Antonio Sánchez...– for their support and help while I have been working on the Ph.D. Thesis. Other colleagues outside of LSD have also helped me with some tasks; Daniel Lebrero gave me some useful comments in a first sight to some chapters; Carlos Escudero provided me access to some technical reports from Gartner I consulted; Jesús Milán helped me with suggestions about the final format of the book. Thank you all!

Of course, I would also like to thank my advisors, Marta Patiño-Martinez and Ricardo Jiménez-Peris, for providing me the topic and the environment for completing this Ph.D. Thesis and being the effective authors of the 50% of the effort.

Google, DBLP, CiteSeer and other information sources, all the software and hardware I have used to develop the system prototypes, experiments and also this book, have also had an important role in materializing this work.

Finally, I would like to thank all the people I have missed and all the things and opportunities I have slipped through my hands during the time this task has last. Has it been worth the effort? Who the f@#% knows?

Francisco Pérez-Sorrosal

Madrid, March 2009

Abstract

Information systems constitute the backbone of modern companies. They support e-business applications and contain critical data on which depends the work of many people in all the levels of organizations. More precisely, the middleware that supports multi-tier and service-oriented applications is nowadays an important cornerstone for many businesses. There are many applications and services that require transactions running on top of this middleware. Transactions are used to ensure consistency of critical data in the presence of concurrent accesses and failures. High availability and scalability are also two desirable properties for the middleware platforms where the applications are deployed. When a failure occurs in a system, clients may not access the applications deployed on it, compromising their availability. Thus, high availability is a key factor for applications because they enable e-businesses to run in a 24/7 fashion. On the other hand, a system is said to be scalable if its performance increases after adding more resources to it. Scalability indicates the capacity of a system of handling growing amounts of work. Thus, the inclusion of these properties in current information systems can make the difference between a competitive and a non-competitive organization.

Providing high availability and scalability to current middleware platforms is not trivial. Fist of all, the inclusion should be transparent for the applications. This requires that the underlying middleware infrastructure provides and combines the adequate techniques, such as replication, failover and recovery. Moreover, data consistency of stateful applications must be guaranteed. Finally, when trying to get one of these two properties (e.g. high availability), the other (scalability) may be compromised and vice-versa. Most of the middleware platforms that claim to provide these properties for stateful applications do not provide both features at the same time, presenting trade-offs. To the best of our knowledge, when a middleware platform claims to support both features, it only supports to scale out stateless applications. Thus, new techniques and technology to overcome these trade-offs must be investigated.

This Ph.D. Thesis aims to improve current middleware platforms. Our contributions allow to provide transparent high availability and scalability to stateful transactional applications and services. We have developed a set of replication protocols and services at middleware level, which provide these two properties to multi-tier and service-oriented architectures (SOAs). Our protocols also guarantee the consistency of stateful applications and services running in the replicated infrastructure. Moreover, node failures are transparent to the clients.

The replication protocols for multi-tier architectures are suitable for stateful applications running in clusters of application servers connected through LANs. In order to guarantee high availability, we have also developed recovery protocols that allow to recover failed nodes or add fresh ones to the cluster without stopping the system (online recovery).

With regard to SOAs, we have developed a replication framework that provides high availability for critical web services. The web service replicas can be located in different nodes of LANs or WANs. The framework respects the web services autonomy, so the replicated web service is accessed in the same manner as the non-replicated one. Moreover, the framework allows to easily deploy the web service replicas from the original non-replicated web service.

Resumen

Los sistemas informáticos constituyen la columna vertebral de la empresa moderna. Se encargan de ejecutar aplicaciones clave para los negocios y de almacenar datos críticos de los que depende el trabajo de muchas personas. Se puede decir que el middleware del que dependen las aplicaciones multicapa y orientadas a servicios es hoy en día clave para muchos negocios. Sobre este middleware se ejecutan miles de aplicaciones y servicios transaccionales. Las transacciones se utilizan para garantizar la consistencia de datos criticos ante accesos concurrentes y fallos. Otras dos propiedades interensates para el middleware son la alta disponibilidad y la escalabilidad. Cuando ocurren un fallo en el sistema, los clientes pueden quedarse sin acceso a las aplicaciones desplegadas en él, comprometiendo su disponibilidad. Por ello, la alta disponibilidad es un factor clave para los sistemas, ya que posibilita ofrecer servicios 24/7. Por otro lado, se dice que un sistema es escalable, si su rendimiento aumenta cuando se le añaden más recursos. El que un sistema sea escalable indica su capacidad para gestionar cargas de trabajo crecientes. Por lo tanto, disponer de alta disponibilidad y escalabilidad en los sistemas de información puede marcar la diferencia entre una empresa competitiva y otra que no lo sea.

Incluir alta disponibilidad y escalabilidad en las plataformas middleware actuales no es trivial. Por lo pronto, los mecanismos que las proporcionen deberian ser transparentes a las aplicaciones. Esto requiere que el middleware proporcione y combine adecuadamente técnicas tales como replicación, gestión de fallos y recuperación, garantizando al mismo tiempo la consistencia del estado de las aplicaciones. Finalmente, también hay que tener en cuenta que al intentar obtener una de estas dos propiedades (p.ej. alta disponibilidad), la otra (escalabilidad) puede ser comprometida y viceversa. Hasta donde sabemos, la mayoría de las plataformas middleware existentes que dicen ofrecer alta disponibilidad y escalabilidad, o bien no proporcionan ambas características al mismo tiempo (comprometiendo alguna de ellas) o bien las proporcionan sólo para aplicaciones que no mantienen estado. Por lo tanto, para superar estas limitaciones se deben investigar nuevas técnicas y aplicarlas a las plataformas existentes.

Esta tesis pretende contribuir a la mejora de las plataformas middleware actuales. Nuestras contribuciones ofrecen, de manera transparente, alta disponibilidad y escalabilidad a aplicaciones transaccionales y servicios con estado. Hemos desarrollado un conjunto de protocolos y servicios de replicación a nivel de middleware que proporcionan estas dos propiedades en arquitecturas multicapa y orientadas a servicios. Nuestro trabajo también garantiza la consistencia del estado de las aplicaciones y servicios que se ejecutan en la infrastructura replicada. Además, los fallos de los nodos son transparentes a los clientes.

Los protocolos de replicación para arquitecturas multicapa estan orientados a aplicaciones con estado que necesitan alta disponibilidad y escalabilidad y que se ejecutan en clusters conectados internamente a través de redes locales. Para garantizar alta disponibilidad, también hemos desarrollado protocolos de recuperación que permiten recuperar nodos que fallaron o añadir otros nuevos al cluster sin detener el funcionamiento del sistema.

En cuanto a las arquitecturas orientadas a servicios, hemos desarrollado un framework de replicación que ofrece alta disponibilidad para servicios web críticos. Las réplicas de dichos

servicios pueden estar localizadas tanto en redes de área local como de área extendida. El framework respeta la autonomía de los servicios web. Esto significa que el acceso a un servicio web replicado se realiza de igual manera que al servicio sin replicar. Además, el framework permite desplegar fácilmente réplicas del servicio web a partir del servicio web original.

Middleware for High-Available and Scalable
Multi-Tier and Service-Oriented Architectures

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The Wolf: *You're... Jimmie, right? This is your house?*
Jimmie: *Sure is.*
The Wolf: *I'm Winston Wolfe. I solve problems.*
Jimmie: *Good, we got one.*
The Wolf: *So I heard. May I come in?*
Jimmie: *Uh, yeah, please do.*

– Quentin Tarantino (*Pulp Fiction*)

Nowadays, in a society fully integrated with the Internet and the World Wide Web, the requirements that both, users and enterprises need from current information systems and applications, are considerably different from those of the early years of computer science. High availability and scalability are two important non-functional requirements to take into account when building today's advanced system architectures and applications. The inclusion of these requirements in current information systems can make the difference between a competitive and a non-competitive organization. This Ph.D. Thesis aims to contribute to the provision of these two valuable features in current software architectures whilst preserving the consistency of the data required by applications and services.

1.1. Overview

The following sections contextualize this Ph.D. Thesis and present its goals. The chapter is organized as follows; Section 1.2 motivates the work done in the thesis. Then, the main

concepts of the Ph.D. Thesis are presented: middleware (Section 1.3), distributed computing and clustering (Section 1.4) and the role of high availability, scalability and consistency in the context of current information systems (Sections 1.5, 1.6 and 1.7 respectively). The software architectures used in this thesis –that is multi-tier and service-oriented– are described in sections 1.8 and 1.9. Once the context of the thesis has been defined, Section 1.10 details the goals of the Ph.D. Thesis. Finally, Section 1.11 describes the organization and contents of the following chapters.

1.2. Current Trends in Software Architectures and Applications

Computer systems and applications are the basis for the smooth running of the businesses of modern companies. The work and productivity of many people depend on the business applications and services deployed in the information systems of the companies, as well as on external applications and services provided by third-party organizations. Nowadays, companies offer their services to their clients through the Internet by means of web-based applications and web services (e.g. Amazon, eBay, bwin...). These web services can be composed, and many companies are trying to take advantage of the service compositions to provide new functionalities to the users. One of the latest trends in the industry is to offer/rent the hardware and software infrastructures of a company as a service through Internet (the "cloud"), what is called *software as a service* (SaaS). That is, the companies allow their clients to deploy remotely their own custom-made applications in the technology platforms of the company, without worrying about the maintenance of the underlying infrastructure. Examples of these new services are Amazon Web Services [Ama] and Google Application Engine [Goo]).

All these applications and services manage critical data for the companies, so it is crucial to guarantee that these applications and data are available, up-to-date, consistent and safe. The success of many businesses relies on the access to the precise information at the right moment, even if a part of the system fails. If an application fails and becomes unavailable for a certain period of time, that would mean a real problem for a company in terms of time and money. However, the computer systems and the *middleware* where the applications and services are running are not completely reliable. Hardware components may break down and software is susceptible to failures in the advent of unplanned events or situations. In order to deal with these unexpected behaviors and provide high availability to both, the computer systems and the middleware, fault-tolerance is used. A fault-tolerant information system is reliable and flexible enough to guarantee the demands of the companies regarding to the *high availability* of their applications and services. Thus, high available architectures are one of the key concepts needed to deploy competitive and valuable applications and services.

In addition to high availability, many mission-critical applications (e.g. those related with banking and billing, air traffic control...) demand strong data *consistency*. Consistency ensures that several simultaneous client operations on the same data do not corrupt the results obtained by the clients. Data consistency in databases has been obtained traditionally by means of transactions [Gra81]. The same concepts are used in many current transactional applications but at a different level of abstraction. The so-called stateful applications keep persistent data retrieved from data sources at the middleware level to improve the performance. Transactions allow to maintain consistent the middleware data when concurrent clients access the application. However, the well-known ACID properties of traditional transactions are not

adequate for certain applications that include scenarios with complex transactional interactions (e.g. workflow-based applications). In order to model and implement the requirements of these applications, advanced transaction models [Elm92, JK97] have been proposed.

There are also many web-based applications that must serve an increasing number of client requests due to foreseen or unforeseen events (e.g. they provide a critical functionality or become a very successful service). These kind of applications use to manage large amounts of data and must provide a good performance and response times. Fore example, the so-called *extreme transaction processing* (XTP) applications [PGNS07] require more resources than traditional transactional applications due to the high loads of transactional operations that they must support. However, the underlying infrastructure of information systems may not be prepared to support high workloads. *Scalability* is another desirable feature for information systems that enables to increase the power of computer systems, adapting them to these new requirements. Basically, there are two possible ways to scale a system: scale up (adding more resources to a single node) and scale out (adding more nodes to the system). Traditionally, the easiest ways to achieve scalability is investing money in a more powerful hardware infrastructure. However, this solution may not be either affordable or practical for many companies. Thus, alternative and cheaper solutions to attain scalability through software architectures are being explored.

As applications evolved in complexity, the systems with a single server running monolithic applications became a bottleneck. The next step was to split the functional parts of an application into different layers and distribute them in different nodes. This layered approach constitutes what is called a *multi-tier architecture*. For example, in three-tier architectures, the user interface of the application can be separated from the business logic and from the persistent data stored in a database. Nowadays, many web-based applications are developed and deployed following this approach. However, the underlying infrastructures that run these applications lack of a complete support for high availability, consistency and scalability.

Internet has made business relationships more and more dynamic. Currently, many businesses are crossing the boundaries of the companies, so new application scenarios are arising: de-centralized business operations, business integration of two merging companies, intra-enterprise data distribution to edge nodes to increase performance... The interoperability in these scenarios imply the management of complex interactions among the applications and services of the different parties. This has always been a handicap for application software vendors. *Web services technology* [New02, ACKM04] and *service-oriented architectures* (SOAs) have arisen to simplify the implementation of solutions for these kind of scenarios. In a SOA, the enterprises offer their services through a technology-agnostic standard web interface. So, the services they provide can be accessed through the web without depending on their internal implementation. However, despite the service-oriented approach is being widely adopted by the companies [Can07], there are still open questions and problems about how to implement SOAs, such as how to provide high availability to this new wave of service-based enterprise applications.

Multi-tier and service-oriented architectures are the most common software architectures in current enterprise information systems. The middleware that supports the implementation of applications following the principles of these two kinds of architectures is one of the cornerstones that support many e-businesses. With regard to high availability and scalability in multi-tier architectures, the companies build redundant information systems using *distributed systems*, specially *clusters*. A cluster is a set of computers that work together acting as a single

one. Usually, building a cluster is a cheaper solution than buying a single computer providing comparable performance or availability. In the last few years, cluster-based infrastructures have gained a lot of attention in order to tolerate failures and increase the performance of critical applications. In order to achieve high availability in a distributed environment such as SOA, the critical services can be deployed in redundant nodes connected through LANs or WANs.

This Ph.D Thesis aims to provide high availability and scalability to multi-tier and service-oriented stateful applications requiring strong data consistency. The middleware solutions supporting these applications must be reliable and flexible enough to provide these features in clusters or distributed systems. The next sections extend and contextualize the main concepts related to this Ph.D. Thesis commented above.

1.3. Middleware

Middleware is an intermediate software layer that allows to integrate different software components and applications. Application integration is necessary to bridge the gap among the high amount of isolated applications –such as ERPs, CRMs or SCMs– that companies own. In a company, the data managed by these applications is commonly scattered and duplicated in different enterprise information systems. A lot of middleware technologies for application integration have arisen to fill this gap: data warehouses, enterprise application integration (EAI), business process management (BPM), web applications (portals...), web services...

Platforms such as CORBA [Objb], J2EE/JEE [Sun03b, Sun06a] or .NET[Mica] are examples of middleware that aim to ease application integration. These platforms define a software infrastructure called *application server* and provide a component model to build applications. The application server eases the applications the access to common services (e.g. security, resource location, transactions, object/relational mapping...) and provides them an execution environment. Thus, when implementing an application, this avoids to the developers to re-implement code to integrate external resources and applications.

Web services is another recent middleware-based technology that also aims to ease application integration. A *web service* (WS) is a self-described application component designed to be published in the web and accessed through an standard interface. The functionalities provided by several web services can be integrated in new service-oriented applications. Current application servers include facilities to wrap current applications and components as web services as well as to implement new service-oriented applications.

Thus, application servers are currently a centerpiece in the software infrastructures of enterprise information systems because many multi-tier and service-oriented applications run on top of them. This means that reliable middleware infrastructures based on application servers are required to deal with possible failures.

1.4. Distributed Computing and Clusters

Current applications need to handle a large amount of page updates (e.g. web applications based on web 2.0) and machine-to-machine traffic (e.g. service-oriented business applications). These applications usually need more computing power than a single computer can provide. A possible solution for particular applications, is to purchase a new computer system with a

powerful processor and components. However, this is not a cost-effective solution. Moreover, future technology improvements in processor fabrication will be constrained by the physics.

A more feasible approach to overcome the limitation in computing power of processors is to connect multiple processors and coordinate their computational work. In this way, a complex computational task may be divided into simpler tasks that are assigned to the processors. These kind of systems are called *parallel computer systems*. Massive Parallel Processors (MPP), Symmetric Multi-Processors (SMP) and Multi-Core Processors are examples of parallel computer systems.

Another approach lies in combining the computing power of several independent computer systems to complete together a computational task. This approach is called *distributed computing*. Distributed systems can be considered conventional networks of independent computing systems, called nodes. Thus, a distributed system provides three primary characteristics [Sch93]; (1) it contains more than one physical computer; (2) there are I/O paths interconnecting the computers; (3) the computers cooperate to maintain some shared state.

A *cluster* is a special kind of parallel or distributed computer system that includes a set of stand-alone computer nodes interconnected through some network technology with low communication latency. Figure 1.1 shows a general architecture of a cluster. The nodes are usually connected though a high speed LAN and using fast communication protocols and services[1]. The computer nodes may work collectively as a single computing resource or they can operate as individual computers [BB99]. A middleware layer in the cluster is responsible for offering a unified view of the system, masking the complexity of the internal configuration of the cluster to its clients and applications. Finally, the sequential or parallel applications run on top of the middleware layer.

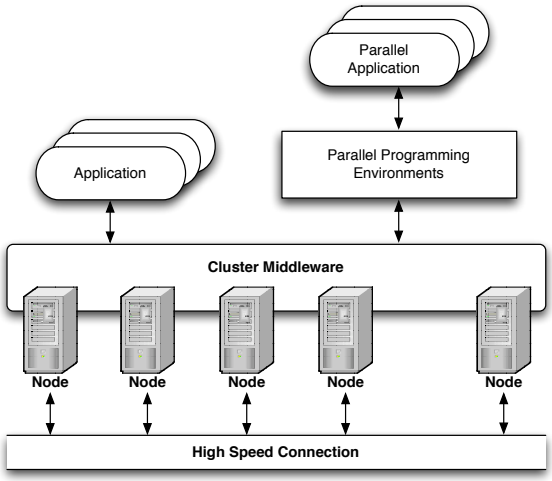


Figure 1.1: General architecture of a cluster

The basic features provided by cluster infrastructures are:

- **Cluster Membership** - This mechanism allows to recognize automatically new cluster members or failures. Group communication systems' facilities can be used to implement

[1]Group communication systems (GCSs) discussed in Chapter 2, Section 2.7 offer some of these services. For example, GCSs provide reliable communication and group membership to help in building communication in cluster infrastructures.

this feature. A heartbeat mechanism can also be used to allow clients/servers to keep track of availability of each other.

- **Farming** - It allows hot deployment of applications across a cluster. This means that when an application is deployed in the cluster, it is automatically deployed in all the members.

- **Load Balancing Mechanism** - It is either a software or hardware component that allows to distribute client requests among the available members of the cluster.

- **Session Management** - Sessions are server-side state application data associated with a particular client. Session data can be accessed by multiple request of the same client. Some clusters include a mechanism called session affinity[2] in order to allow multiple requests for the same session to be routed to the same cluster member.

- **State Replication** - It is a mechanism for preserving application state –either session or persistent– across a cluster. It can be based on the communication facilities provided by a GCS or not. When replicating application state data, the replicated state must be kept consistent in all the members of the cluster.

- **Failover** - A mechanism that allows a cluster member to recover the state of a failed member and continue the processing of requests with no interruption to the end user.

There are different types of clusters. In order to perform a cluster classification, several criteria may be selected [BB99]: node configuration, node ownership, levels of clustering... For example, depending on the node configuration, a cluster may be homogeneous (all nodes have a similar architecture and run the same operating system) or heterogeneous (the nodes neither do not have a common architecture nor run the same operating system). The criterion used in this thesis is the application target. Depending on this criterion, clusters are categorized into *high availability* and *high performance* clusters. It has been shown that many critical applications need to be fail-safe in order to be available when a failure occurs in the underlying system infrastructures. High available clusters allow to provide a high available environment for running critical applications. The performance of applications also improves with the support of scalable software environment. The next sections describe the features of these two kinds of clusters.

Building interconnected systems such as distributed systems and clusters, requires to address several issues [Sch93]:

- **Independent Failures** - As different computers are involved in these kind of systems, the system as a whole must support individual failures and continue working.

- **Unreliable Communication** - Because the interconnections among the different computers may be unavailable or may not be reliable, messages may be lost or corrupted.

- **Insecure Communication** - The interconnections among the computers may be susceptible to eavesdropping and message modifications.

[2] Also known as sticky session.

- **Costly Communication** - The interconnections among the computers usually provide lower bandwidth and higher latency than communications among different processes within a single machine.

Distributed computing and clusters are the pillars for the construction of the high available and scalable multi-tier and service-oriented infrastructures presented in this Ph.D. Thesis, so these issues must be taken into account in the remainder of this work.

1.5. Fault Tolerance and High Availability

Fault tolerance is the ability of a reliable system of being able to maintain or reconstruct a consistent state in the presence of failures [WV01]. The fault tolerance of a system also means to respond gracefully to the failures in its components and continue operating properly. All this implies that a fault-tolerant system must be designed to cope with possible failures of its components without affecting the users and corrupting the state of the system. Failures in components may be classified in two classes:

- **Fail-stop/Crash Failures** - In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops [Sch84, GHM+99].
- **Byzantine Failures** - The failed component exhibits a type of behaviour that is arbitrary and malicious, possibly sending conflicting information to different parts of the system [LSP82].

Fault tolerance in a computer system can be achieved as follows:

- **Building the system with fault-tolerant components** - Fault-tolerant components are components that can continue operating even if one or more subcomponents have failed. Usually this failures cause the overall system throughput or response time to degrade.
- **Having replicated/redundant systems** - A replicated system is composed by a set of clones of the original system –called replicas– that may operate either in parallel or become active when a failure in a replica is detected.

The fault tolerance can be implemented using software or special hardware components. Fault tolerance in this thesis is provided by means of replicated software running in a cluster or in a distributed system. The fault tolerance allows to provide high availability to critical applications and services. High availability is commonly understood as continuous availability. This implies computing systems capable of providing uninterrupted service to their users. In an information system, it is desirable that the components (either hardware or software) are up and running and fully functional for as long as possible. Many services should remain available despite node and connectivity failures to enable business interactions on a 24/7 basis. In the presence of a failure, the system should continue providing the same service to the users despite that the throughput of the system could be affected.

When a failure occurs in a reliable computer system, a *failover mechanism* is triggered. A failover mechanism takes care of masking the failure to the clients of the failed replica by switching over to another non-faulty replica. In clusters of servers, it is also necessary to isolate users and applications from the failures that may occur in the individual replicas. The idea is to give the illusion that they are dealing with a non-replicated system. These kind of mechanisms are called *transparent fault tolerance* mechanisms. After a failure, a fault-tolerant stateful system must launch a recovery process in order to reconstruct a consistent state and continue operating without corrupted data.

In the failure model considered in the thesis, only complete crashes of the replicas running the applications are considered. Moreover, transparent fault tolerance is provided to the clients of a replica. Finally, the replicas should have the *fail-stop* property, in the sense that each replica is shutdown after detecting an error [WV01].

1.5.1. High Availability Clusters

High availability clusters –also known as *failover clusters*– are built with the aim of improving the availability of services and applications running in the cluster. This is achieved by means of redundancy. The objective of introducing redundancy is to eliminate single point of failures. In a non-redundant system, when a particular service crashes, the service is not available until someone fixes the problem and restarts the service. High availability clusters provide redundant nodes that are used to provide the same service or application when other nodes fail. Figure 1.2 shows a possible architecture for a high available cluster. The set of nodes is usually connected through a fast LAN. The applications requiring high availability are deployed in all the nodes of the cluster. In the example, the requests from the clients are redirected only to a unique node: the primary. Upon changes on the state of the applications, the primary node is in charge of sending the new state to the backup nodes. In this way, in case of the failure of the primary node, the rest of the nodes still have the updated state and the clients can be redirected to one of them through a failover mechanism.

This kind of clusters are mainly used for attaining high availability in databases, file systems, web sites... In this thesis, high availability clusters are used to provide high availability to multi-tier applications. Chapter 3 describes a protocol providing these features.

1.6. Scalability

The ability of a system to remain effectively and efficiently operational when adding new resources and users to its infrastructure is called *scalability* [CDK01]. Computer systems are scaled basically to increase their computing power. The importance of having a scalable system is highly significant in enterprise information systems because the computational requirements of the applications are growing day after day (e.g. support more client requests, decrease the response time for the requests, add functional complexity...). Thus the computer systems supporting those applications must be adapted to fulfill their requirements. Basically, there are two approaches for scaling a system [MMSW07]:

- **Scale up** - With regard to hardware, to scale up –or *scale vertically*– a system implies the addition of CPUs and/or memory to a single computer using shared memory. In software,

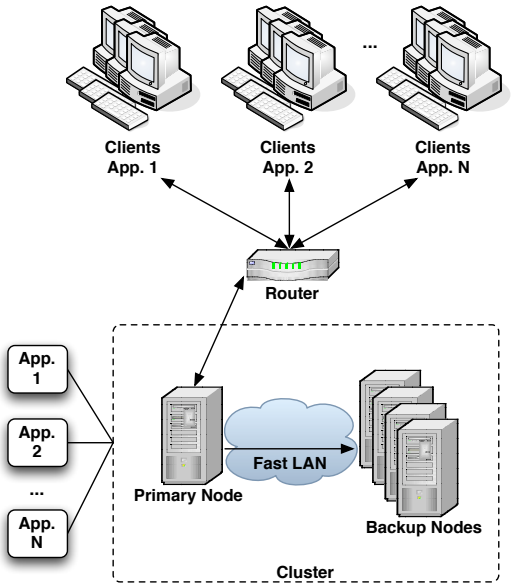


Figure 1.2: High availability cluster/Failover cluster

this can be achieved expanding the number of processes or threads running concurrently in applications (e.g. in the Apache web server, the number of threads ready to serve client requests can be adjusted depending on the requirements of the applications).

- **Scale out** - To scale out –or *scale horizontally*– a system implies to add more nodes to a cluster. The middleware supporting the applications must be ready to provide scalability.

However, both approaches exhibit trade-offs. When scaling-out a system, some applications do not lend themselves to a distributed computing model. Moreover, a large number of computers means to increase the complexity of both, the management tasks as well as the programming model for applications. It is also very important to try to avoid performance bottlenecks (e.g. latency among the nodes, throughput...). However, with regard to the price of the resulting scaled systems, the scale out approach can be a less expensive solution for those applications that fit its paradigm. Apart from the price, another drawback of scale up approach is that –in many cases– the availability of the system is not guaranteed. That is, thee system has to be shutdown in order to carry out the migration process of its running applications.

In this thesis, scalability is understood as the scale out of a cluster by means of the appropriate software architecture.

1.6.1. High Performance Clusters

High performance clusters are sometimes called *load-balancing clusters* or *server farms*. Although they are primarily designed for improved performance, they commonly include high availability features as well. The performance is achieved scaling-out the cluster.

Figure 1.3 shows an example of a high performance cluster configuration. In this configuration, the incoming load injected by the clients is distributed to a group of back-end nodes

connected by a high speed LAN. The load balancer redirects the client requests to the nodes that are less overloaded. When either all the nodes or some of them share the same application state –e.g. if the state has been replicated to provide high availability– it is important to maintain the consistency of the shared state in the required nodes. Moreover, if the replication protocol is designed properly, scalability can be achieved adding more nodes to the cluster, while preserving the high availability and the state consistency of the applications.

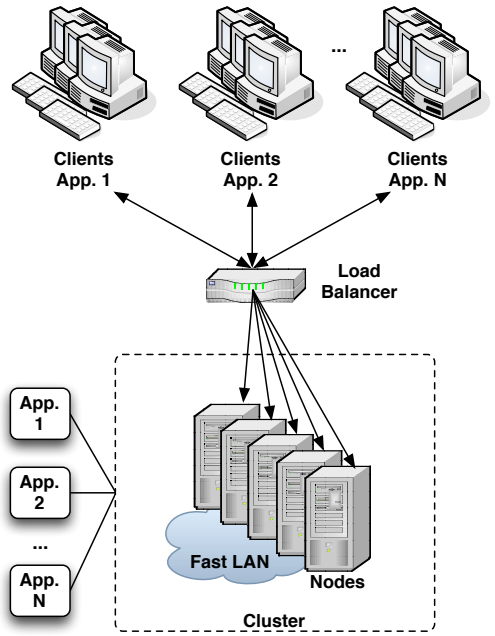


Figure 1.3: High Performance cluster/Load-Balancing cluster/Server farm

Chapter 4 of this Ph.D. Thesis describes a protocol providing high availability and scalability for applications running in a high performance cluster.

1.7. Consistency

Consistency preservation captures the concept of producing states in a system that are meaningful and correct for the applications that use them [BHG87]. For example, let us consider an scenario where a client intends to purchase books through an online bookstore application. In this scenario it is important to keep certain data consistent. When the client is browsing the catalog and selects a book to purchase, the view of the shopping cart must be maintained consistent with the stock of books managed by the provider. Consistency is especially important in applications where concurrent accesses to data may occur (e.g. several clients attempting to buy the same book) or when data are distributed across different computers (e.g. if the bookstore application runs in a cluster).

State consistency for applications is provided by means of transactions. From the point of view of an application, a transaction is essentially a program execution that reads and modifies one or more data sources (e.g. database systems). Transactions provide several system-guaranteed properties that simplify the development of applications. In particular, applications

are freed up from taking care of the issues of [WV01]:

- **Concurrency** - that is, the effects that may result from simultaneous or even parallel executions and data accesses.

- **Failures** - that is, the effects of interrupted executions because of process or computer failures.

The increasing degree of sophistication and complexity of some modern business applications is also demanding support for flexible long running transactional activities. Due to the long running nature of certain business activities, some of the properties of traditional transactions are not adequate for them. The so-called advanced transactional models [Elm92, JK97] where developed in order to support the requirements of these new applications. It is also crucial for the applications based on advanced transaction models to guarantee data consistency.

The software elements (components, web services...) that conform the modern transactional applications can store two kinds of application state that must be preserved consistently:

- **Session state** - It is application state related to a particular client (e.g. by means of a client id) that is valid during a limited amount of time (e.g. several invocations). The components that store this kind of state are called *session components*. As these components are tied to a particular client, they are not accessed concurrently.

- **Persistent state** - It is application data that has been retrieved from a data source (e.g. a database). The components where this kind of state is stored are called *persistent components*. Persistent components may be accessed by multiple clients concurrently.

The decomposition of an application in different layers (e.g. web-based applications) also makes more difficult the achievement of consistency. In these cases, consistency must be guaranteed not only for the individual components or services, but also across the different parts of the application.

Moreover, in applications deployed in clusters or services deployed in redundant nodes, consistency must be preserved among all the individual replicas running the application or service instances, providing a single consistent view of the system to the clients. In replicated transactional information systems, the most restrictive correction criterion is called *one-copy serializability* [BHG87, WV01]. For this criterion, any replicated object, component or data must appear as a single logical copy in the system despite the existence of several copies (*one-copy equivalence*), and the concurrent execution of transactions on the different copies must coordinated in such a way that is equivalent to a sequential execution on the logical copy (*serializability*). Moreover, transactional atomicity in such systems must ensure that a successful transaction commits in all the replicas. On the contrary, if the transaction fails, it must be aborted also in all the replicas. Both, one-copy serializability and atomicity are desirable features for a replicated system. However it is hard to implement an efficient replication protocol with these features, so real systems must implement weaker correction criteria such as *one-copy snapshot isolation* for the sake of performance.

The Ph.D. Thesis addresses these scenarios providing the necessary mechanisms to enable consistency in high available and scalable multi-tier and service-oriented architectures.

1.8. Multi-Tier Architectures

Multi-tier architectures –also known as n-tier architectures– are widely used in current enterprise information systems. Many web-based applications are built using this approach. A multi-tier architecture is a client/server-based infrastructure in which an application is split in different layers (tiers), each one providing well-defined interfaces that are executed by distinct software elements. The three-tier architecture is a well-known example of this kind of architectures. In a three-tier architecture, the application is split in: *presentation tier*, *application tier* and *data tier*. The presentation tier represents the user interface of the application. In the application tier, also known as middle, business or logic tier, it is built the logic part of the application. Finally, the data or back-end tier allows the access to the persistent information required by the application. Each tier can be deployed in the same centralized node or in different nodes connected through a network. Figure 1.4 shows an example of a three-tier architecture in which each tier is distributed in a different node.

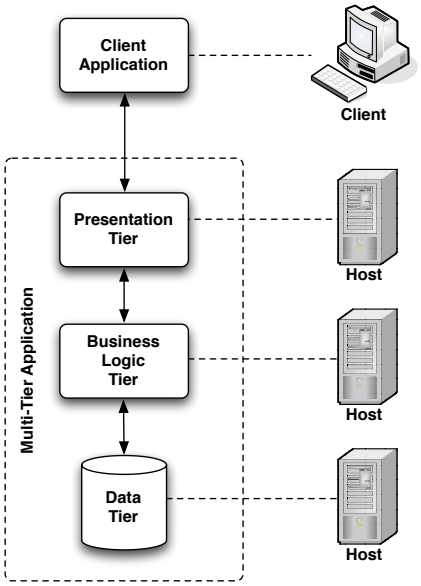


Figure 1.4: Example of a multi-tier architecture (Three tier)

As a modular software, one of the great advantages of multi-tier architectures is that tiers can be upgraded or replaced independently as requirements or technology evolve. Another advantage is that they can be scaled if they are properly implemented. For example, it is possible to balance and distribute the workload among multiple (and often redundant) server nodes deployed the same tier. In this way, since most of the workload is processed simultaneously, the overall system performance can be improved.

Of course, multi-tier architectures also have disadvantages. Many people claim that it is more difficult to program applications. However, there are programming frameworks that provide an easier way to implement, deploy and test applications than performing all those actions simply "by hand". Another disadvantage is that it increases the network traffic due to communication between tiers if each tier is deployed in a different node. However, by selecting the right distribution of the tiers in the system infrastructure, a trade-off solution between

performance and network traffic can be achieved.

There are different frameworks that allow the construction of multi-tier architectures (e.g. Ruby on Rails, CORBA, .NET, J2EE/JEE...). Nowadays, the most used in the enterprise is J2EE/JEE [Can07].

1.8.1. High Availability and Scalability in Multi-tier Architectures

The inclusion of high availability and scalability in enterprise information systems based on multi-tier architectures is having a lot of demand. Many recent enterprise applications have been developed following a multi-tier approach. High availability of enterprise applications is crucial for a lot of current businesses. Gartner Group shows in a recent report that one of the main factors in order to decide which application infrastructure to buy for business application projects is its availability features [NPT+07]. Many software vendors selling application servers and other middleware solutions (Oracle, IBM, BEA...) are introducing or improving the availability of their systems. Some of the common techniques for attaining high availability consist in the use of clusters. A high available solution must provide consistent state for stateful applications running in these clustered environments.

However, high availability is not the only one key concept to take into account when a company buys and deploys an infrastructure for running its applications. The hardware and software infrastructures where the business applications run, should be able to scale in order to absorb a possible growth of turnover. Investing a lot of money in a new information system can be problematic for a company with an e-business that grows unexpectedly and its current information system can not deal with higher loads because it is not able to scale. Today's large scale information systems, such as those related with search engines (Google, Ask, Yahoo...) or large databases (Wikipedia, IMDB...), need more and more scalability. The same applies for small or medium size companies that expect a growth in their businesses or big companies that must improve the response times of certain services or applications. It is neither affordable nor recommendable for them, to change their information systems overnight. When they plan to buy or build an enterprise information system, it is better for them to invest in choosing a system able to scale that to save money buying a system that only satisfies the current (or maybe short term) requirements. In the long term, it is for sure that this fact will save money for the company.

1.9. Service-Oriented Architectures

Currently, there is a lot of controversy about what a "service-oriented architecture" is [WM06]. However, in a simple way, a service-oriented architecture can be defined as a **technology-agnostic collection of services that communicate with each other**. A *service* is a software component that provides a well-defined and self-contained functionality and that does not depend on the context or state of other services. That is, a service is *loosely-coupled* to other services. Services can be combined to provide new functionality through the so-called *service compositions*. The communication between services can involve either simple data passing or it could involve two or more services coordinating some activity.

Therefore, in SOAs, some means of connecting services to each other is needed. As SOAs are technology agnostic, this means that communication may be implemented with any

communication protocol (RPC, DCOM, CORBA, HTTP, SOAP...). Component models such as Microsoft's DCOM, CORBA or more recently EJB could be considered as examples of SOAs. The main difference between these technologies and SOAs relies on that the SOA services interoperate based only on a contract that is independent of the underlying technology platform or programming language. However, CORBA and EJB components can not communicate directly with each other.

When a service is accessed through the web, it is called a *web service*. A web service is a self-described application component designed in order to be published in the web and accessed through an standard interface. Web services can be implemented following two main architectural styles: REST [Fie00] and SOAP [W3Ca]. In this thesis, the term "web services" refers to SOAP-based web services. The architecture of SOAP-based web services is defined by standard specifications developed jointly by the most important software vendors such as Microsoft, IBM, Oracle, BEA.... All these specifications are described using the XML language[3]. The basic specifications of the SOAP-based web services are WSDL, SOAP and UDDI [New02]. WSDL [W3C01] allows the description of web services interfaces, that is, their contracts. SOAP [W3Ca] provides the communication layer based on XML to access the web services. Web services engines such as Axis and Axis2 [Apac, Apad] are middleware that enables SOAP-based communication in SOAs. Finally, publication and discovery of web services is made through UDDI [OASb] repositories. UDDI repositories act as a directory that is consulted in order to find a web service. Besides these basic specifications, several web services specifications (also known as WS-*) have arisen in order to solve the different issues that service-oriented applications have brought with themselves (e.g. security, reliable communication, transactions...).

During the last few years, the success of web services has relaunched the interest for SOAs. Using service orientation in the architectural design facilitates reusability, flexibility, interoperability and agility of applications. Service-oriented applications are nowadays a new choice to build enterprise applications. Figure 1.5 shows an example of a service-oriented application. In the example, the service-oriented application ($SO - App$) requires access to four web services (WSA, WSB, WSC and WSD) in order to fulfill its functionality (service composition). The dotted curves represent different organizational boundaries. Web services A and B are located in nodes belonging to the same organization as the composite service, so they may be accessed through the LAN of the organization. However, web services C and D are provided by external organizations. Thus, when the SOBA accesses these services the communication takes place through a WAN (e.g. Internet). Thanks to the WSDL interface of web services A, B, C and D, they provide a standard access to the functionality that is independent of their concrete implementation. The SOAP protocol allows to solve the problems of heterogeneous and incompatible communication protocols that arise in composite applications involving different organizations.

With regard to the implementation of a SOA, the web services offered by organizations can be integrated into a new service tier. Figure 1.6 shows an example of how this integration can be implemented. Firstly, the example shows how the web services layer abstracts the different computing resources of the organization (software components, databases, files...), offering

[3]XML is a language similar to the well-know HTML. However, XML is richer and flexible than HTML. It can be used not only for document presentation (as HTML) but for storage and communication of data too. XML uses a flexible tagged structure that makes it more robust than a fixed record format for communication.

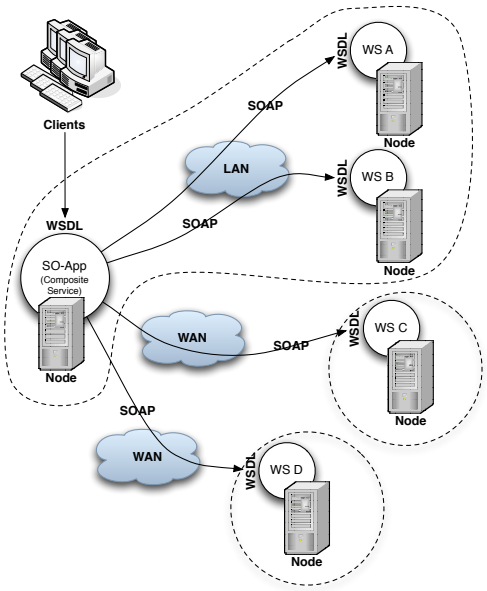


Figure 1.5: An example of a service-oriented application

a unified view of them as services. In the left side of the figure, it is shown the integration of a web services with an existing multi-tier application. In this case, the business tier offers its functionality through the web service A. In the right side of the figure, two resources of the organization are presented as web services. Web service B abstracts the access to the information included in a file and web service C abstracts the access to the information stored in a database.

1.9.1. High Availability in Service-Oriented Architectures

In a recent report [Can07], Gartner Group made an analysis of adoption of SOA, web services and web 2.0 technologies during the years 2005 and 2006. The report states that SOA and web services are being adopted by the enterprises and entering the mainstream of business applications.

This increasing adoption and success of web services technology for SOA in enterprises is demanding high available infrastructures. In the next years, a high number of mission-critical applications will be deployed in SOAs with web services. The availability of those systems must be guaranteed in case of failures and network disconnections. As examples of web services for which availability will be a crucial issue are UDDI repositories or those services belonging to coordination web service infrastructure, such as web services for security, coordination or transactions. The unavailability of a coordination service impacts the availability of all the partners in the business process. These services should remain available despite site and connectivity failures to enable business interactions on a 24/7 basis.

As in multi-tier architectures, the techniques used for attaining high availability in SOAs rely on the use of service replicas. However, the scenario where the service replicas are deployed can be different. Web services are not only tied to intra-enterprise systems. Web services can

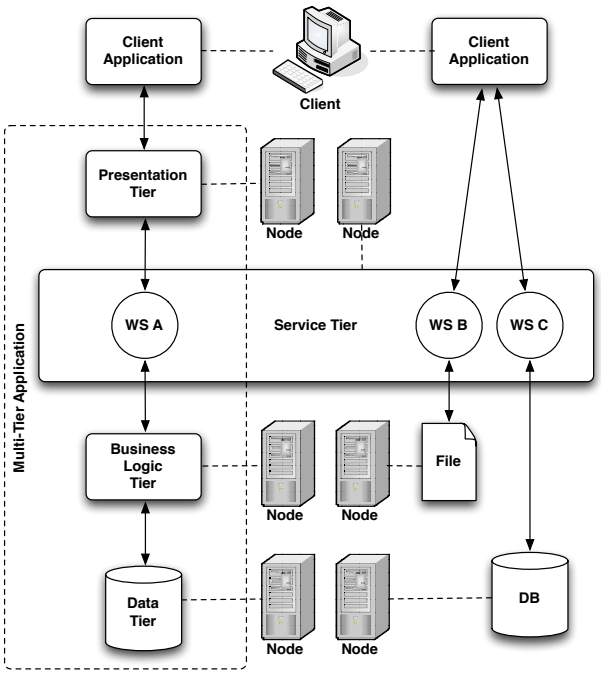


Figure 1.6: Service tier integration in a multi-tier architecture

cross the boundaries of the enterprise and can be composed and connected using wide area networks (WAN). This means that for achieving dependability in SOAs the problems that arise in these scenarios (such a network disconnections) must be fixed.

1.10. Ph.D. Thesis Goals

The overview of the current trends of system architectures and applications presented in the previous sections has highlighted the requirements needed when developing today's applications and services. These requirements are mainly related to the provision of high availability and scalability in the underlying system infrastructures. These infrastructures must also guarantee the consistency of the stateful applications and services running on top of them. In order to provide these features, middleware platforms suitable for clusters and distributed systems are required.

The work done in this Ph.D. Thesis focuses on the area of middleware replication for components and services. The main goal we aim to achieve is to develop a set of replication protocols at middleware level, which provide high availability and scalability in a consistent way to multi-tier and service-oriented architectures. The protocols for multi-tier architectures must be suitable for stateful applications running in clusters of application servers connected through LANs. The protocols for service-oriented architectures must be suitable for critical web services deployed in groups of replicas in LAN or WAN environments. When a failure occurs in one of the replicas, the clients must not be aware of it and the system must continue processing the client requests. In order to achieve this requirement, the protocols must provide

a transparent fault tolerance mechanism. Additionally, for multi-tier architectures, we want to develop a recovery protocol, which must allow both, to recover failed replicas and incorporate new ones to a cluster without stopping the system (online recovery).

1.10.1. Contributions to Replicated Multi-Tier Architectures

The specific contributions to replicated multi-tier architectures have been the following:

- Development of a set of three replication protocols to provide high availability for multi-tier stateful applications deployed in clusters of J(2)EE application servers. Chapter 3 describes this work in deep. This work has been also published in [PPJV06].
  - The state of the applications is maintained consistent in all the replicas of the cluster.
  - The protocols provide high available transactions.
  - The replication and failover processes are transparent to the clients of the applications.
  - The protocols take into account several client interaction patterns (1 request/1 transaction, N requests/1 transaction, 1 requests/N transactions and N requests/M transactions).
- Development of a replication protocol that, in addition to high availability, adds scale out capabilities to stateful applications running in J(2)EE application server clusters. This replication protocol is described in Chapter 4 and has been published in [PPJK07].
- Development of a recovery protocol to recover/incorporate replicas to J(2)EE application server clusters. The recovery protocol is also described in Chapter 4. This work has been submitted for publication to the VLDB Journal.
  - The recovery process is done without stopping the system processing (online recovery). This way, whilst the recovery process is done in the recovering replica, client requests are also being processed.
  - When a failed replica is recovered or new replicas are added to the cluster, the state of the recovering replica at the end of the process will be consistent with the other replicas in the cluster.

1.10.2. Contributions to Replicated Service-Oriented Architectures

The specific contributions to service-oriented architectures have been the following:

- Development of a generic replication framework and a protocol to provide high availability to critical web services deployed in service-oriented architectures. The critical web services are deployed in a set of replicas that can be connected either through LANs or WANs. The infrastructure and the protocol are described in Chapter 5 and the results have been published in [SPPJ06].

- The infrastructure allows to deploy a standard web service as a replicated one without modifying the web service.
- The replicated state of each web service is maintained consistent in all the system replicas.
- The replication and failover processes are transparent to the web services consumers (clients).
- The infrastructure preserves the interoperability requirements imposed by web services technology.

The protocols have been implemented, deployed and tested in application servers supporting both, J(2)EE[4] and web services technologies. Then, the throughput and scalability of the replicated systems have been evaluated. Finally, in order to analyze the feasibility of the protocols, the results of the evaluations have been compared against the equivalent systems without the replication protocols in the required scenarios.

1.11. Outline

The remainder of this Ph.D. Thesis is structured as follows. First of all, Chapter 2 introduces some basic concepts and background related to this work. The following three chapters constitute the core of the thesis. Every chapter presents a specific protocol and its evaluation. Chapter 3 presents the transaction-aware replication protocols to provide high availability to multi-tier architectures. Then, Chapter 4 describes the replication and recovery protocols to provide consistency, high availability and scalability also to multi-tier architectures. To conclude the core part of the thesis, Chapter 5 describes the high availability framework for service-oriented architectures. The framework is used to implement a replication protocol for web services. Then, Chapter 6 compares the related work with the work done in this thesis. Finally, Chapter 7 concludes this Ph.D. Thesis.

[4]Up to version 5, Java Enterprise Edition (JEE) was formerly known as Java 2 Platform, Enterprise Edition (J2EE). From this point on, the acronym J(2)EE is going to be used in order to refer to both, J2EE and JEE. If it is necessary to distinguish between them in the text, each concrete name will be used when required.

CHAPTER 2

Background

You can know the name of a bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird... So let's look at the bird and see what it's doing -that's what counts. I learned very early the difference between knowing the name of something and knowing something.

– Richard Feynman

As it has been stated the previous chapter, the main focus of this Ph.D. Thesis is to provide solutions to achieve consistency, high availability and scalability in multi-tier and service-oriented architectures. In this challenge converge a wide range of research areas and technologies. To better understand the remainder of this Ph.D. Thesis, this chapter compiles the basic concepts and background of the related research areas. When necessary, the middleware technologies used for implementing the solutions proposed in the next chapters are also described.

2.1. Overview

Application servers constitute the main technology that supports the work done in the thesis. So, first of all, the chapter gives some background on application servers in Section 2.2, focussing on J(2)EE technology. Transactions are one of the key concepts in this thesis, so

Section 2.3 describes the basic concepts and how they fit in the thesis. The other key concepts, replication and recovery, are covered by sections 2.4 and 2.5 respectively. Section 2.6 presents the role of caches in middleware. Finally, Section 2.7 describes the most relevant features of the group communication systems used by the protocols presented in the next chapters.

2.2. Application Servers

In the last few years, a lot of web-based applications have been designed, constructed and deployed on multi-tier architectures based on application servers. Application servers are the middleware on top of which have been implemented and evaluated the replication protocols presented in this thesis. An application server provides a common infrastructure –in terms of services, tools and management facilities– to applications deployed on top of it. Some of the services that application servers provide are: a well-defined component model for implementing the business logic of the applications, resource binding and location, transaction management, security and simple data source access configuration. Using these common services and tools, the application developers do not have to re-implement code related to resource management when developing applications, but just focus on the application logic. Additionally, application servers also provide management tools to monitor and control the resources provided to applications (security policies, database connection pools, transaction isolation levels, timeouts...).

Clustering facilities are being incorporated gradually to application servers. Application server vendors implement these facilities in a different way, but all of them look for the best possible cluster architecture that provide high availability for mission-critical applications and scalability for large scale applications.

Service-oriented architectures (SOAs) are also starting to be widely adopted and supported in new enterprise information systems (EIS). In order to implement SOAs, there are some trends in the industry that claim for new programming models and environments far from application servers [PN07]. The reasons argued are that application servers were designed to support web applications and the incorporation of SOA and event-processing capabilities complicates even more an already-complex programming model. However, in the last few years, software vendors such as IBM, BEA, Oracle, Red Hat... have invested a lot of money adapting their application servers to incorporate the required functionality to build SOAs. Thus, SOAP engines and web services tools can be found in the most important application servers both, commercial and open-source.

In Chapter 1 it has been shown that multi-tier and service-oriented architectures do not depend on a specific underlying technology in order to be implemented. In this Ph.D. Thesis, J(2)EE has been used as the reference platform for implementing the protocols for multi-tier architectures. With regard to SOAs, the web services technology based on SOAP has been chosen as the reference platform. JBoss [Reda] and JOnAS [Objf], the application servers used for implementing the protocols of this Ph.D. Thesis, include support for J(2)EE and web services.

2.2.1. J(2)EE Application Servers

The J(2)EE [Sun03b, Sun06a] specification offers an extensible framework for developing distributed web-enabled multi-tiered applications based on the Java programming language [Sunb]. The J(2)EE specification is in fact a set of specifications that include the definition of dynamic web page generation systems –Servlets, Java Server Pages (JSP) and Java Server Faces (JSF)–, a distributed component model called *Enterprise Java Beans* (EJB) and many other useful services for applications such as security, communication or transactions. Every J(2)EE service specification provides a well-defined API to access the service functionality. The concrete implementations of these specifications are called J(2)EE application servers. J(2)EE application servers are the middleware platforms where J(2)EE-based applications are deployed. The J(2)EE specification enables software vendors to build compatible application servers. In this way, the same application can be deployed in multiple application servers without changing the source code. The J(2)EE specification does not prescribe nor states any constraint about clustering. The clustering solutions offered by J(2)EE-compatible application servers are specific of each vendor.

The overall view of the architecture of a J(2)EE application server is shown in Figure 2.1. Most J(2)EE applications follow a three-tier architecture. They consist of a set of web components (JSPs and Servlets) that model the presentation tier and a set of components that model the business logic (EJBs) and access the application data, usually stored in a relational database. Each kind of component is deployed into a specific container (Web or EJB) that creates the component instances. When required, the components can access the services provided by the application server. With regard to the thesis, it is important to describe the EJB component model that allows to implement the business logic of applications. Other sections describe when required other features or services provided by the J(2)EE specification.

The *EJB container* holds and manages the EJB components and is the centerpiece of any J(2)EE application server. The container provides facilities to create, use and destroy instances of EJB components in memory. Furthermore, it provides concurrency control mechanisms to avoid that concurrent client requests corrupt the state of components. There are three types of EJBs: *session beans* (SBs), *entity beans* (EBs) and *message driven beans* (MDB). The business logic of applications is implemented in SBs. The lifetime of a SB is bounded to the lifetime of its clients (their state is volatile and may be reused by different client invocations). Session beans are further classified as stateless (SLSBs) and stateful (SFSBs). SLSBs do not keep any state across method invocations. SFSBs may keep state across invocations of the same client, what is called conversational state or session state (See Chapter 1, Section 1.7). EBs provide in-memory representation of data stored in persistent back-end storages, e.g. databases. The container may use a persistence mechanism in order to load/store the state of EB components from/to a persistent storage. The state of EBs can be persisted in secondary storage either explicitly by the developer (*bean-managed persistency*), or delegating the storage to the container itself (*container-managed persistency*). When using container-managed persistency, the container takes care of reading/writing data from/to the database by generating automatically the adequate SQL statements. Then, the EB components act as a cache for the retrieved data. The object-relational mapper (O/R mapper) is the component of the application server in charge of fetching the data stored in the database into the EB components and vice-versa using database connections. Database connections are established through Java Database Connectivity (JDBC) drivers [Sund]. JDBC defines a standard API

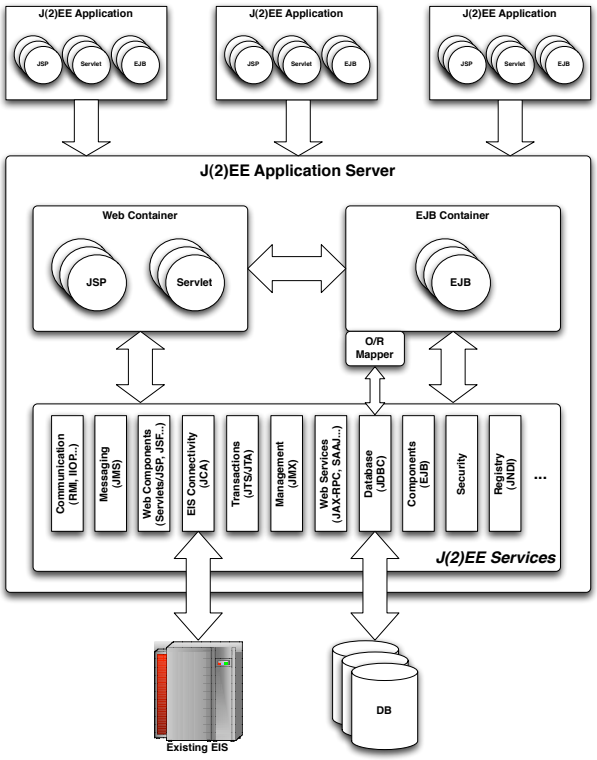


Figure 2.1: Main J(2)EE components and services

to allow independent connectivity between the Java programming language and a wide range of databases. It is the Java version of the well-know ODBC [Micc] in Windows platforms[1]. Finally, the last type of EJB components are MDBs. MDBs are stateless components used in asynchronous communications (e.g. by means of message queues).

In the context of the Ph.D. Thesis it is important to keep consistent the state of applications, so only SFSBs and EBs need to be considered when developing the replication protocols. Moreover, the protocols make use of the container-managed persistency mechanisms provided by the container.

2.3. Transactions

The concept of transactions is a crosscutting concept in the all the core chapters of this Ph.D. Thesis. Transactions have been traditionally related with the database area. However, transaction processing is also very important in middleware-based systems.

In information systems, transaction processing aims to preserve the integrity of a system in a known consistent state. A *transaction* can be defined as a sequence of read/write operations executed on a set of data. Originally, in his classic paper, Gray [Gra81] described three properties related to transactions: *atomicity*, *consistency* and *durability*. Other property,

[1]Microsoft's Open Database Connectivity (ODBC) is a middleware that allows Windows applications to connect to different database systems abstracting the differences of each one of them.

isolation, comes to scene when talking about concurrent transactions. Härder and Reuter [HR83] introduced the *ACID* mnemonic to refer to these well-know transaction properties in 1983. The following is a description of the ACID properties:

- **Atomicity** - It can be resumed with the phrase "all or nothing". It means that either all the operations performed inside the transaction are completed or none is done. Transactions have only two possible outcomes: *commit* and *abort/rollback*. When a transaction commits, all the changes produced in data are persisted. On the other hand, when a transaction aborts, the changes produced in data are discarded.

- **Consistency** - It ensures that a transaction performs a correct transformation of the data, from a consistent state to another. A transaction must not violate any consistency constraint associated with the state. A consistency constraint is a predicate on data which serves as a precondition, post-condition, and transformation condition on any transaction [GR93].

- **Isolation** - It guarantees that, when several transactions are executed concurrently, it appears to each transaction T, that the other transactions were executed either before T or after T, but not both. This means that the changes done by a transaction T are not visible to other transactions until T is committed. The safeguards used to prevent conflicts between concurrent transactions are known as *isolation levels*.

- **Durability** - This property ensures that the changes performed by committed transactions survive failures.

The so-called *transaction manager* (TM) –also called *transaction processing (TP) system/monitor*– is the element in charge of managing transactions in transactional information systems, such as database management systems (DBMSs) [BN96, GR93]. Transaction managers guarantee that the ACID properties of transactions are preserved.

Isolation is the property that guarantees the correct execution of concurrent transactions. An important concept to understand isolation is *serializability*. An execution of transactions is serializable when the result obtained is the same whether the transactions are executed in serial order or in an interleaved fashion [BG83]. Transaction managers implement *concurrency control mechanisms* to produce serializable executions of transactions, solving the consistency problems that arise from concurrent accesses to the same data elements (e.g. data records in a database table, objects, or some other data structure). There are several concurrency control mechanisms: locking (semantic, two-phase...), timestamp ordering, multi-version... Concurrency control mechanisms can be classified in *pessimistic* or *optimistic* [CDK01]. Pessimistic approaches (e.g. locking, timestamp ordering) detect conflicts between transactions as each data is accessed. On the other hand, optimistic approaches, allow transactions to operate simultaneously on data, but some of them are aborted when they attempt to commit. Each one of these approaches has benefits and drawbacks. Locking is better for transactions with more write operations than read operations. Timestamp ordering is beneficial for transactions with predominantly read operations. However these pessimistic approaches may cause the delay of transactions or deadlocks due to the lock of resources when conflicts arise. Optimistic approaches are efficient when few conflicts arise. However, if there are many conflicts, the amount of work that is repeated when transactions are re-executed may be too high.

Various degrees of consistency where described by Gray et al. in [GLPT76]. The original degrees of consistency where reinterpreted by the SQL 92 ANSI committee in what are called the *ANSI isolation levels*: *Read Uncommitted*, *Read Committed*, *Repeatable Read* and *Serializable*. These isolation levels where defined in terms of the serializability definition plus three different phenomena called *dirty reads*, *non-repeatable reads* and *phantom reads* which describe prohibited operation subsequences [BBG+95]. Each one describes a different problem related to anomalous behaviour in the serialization of concurrent transactions.

Dirty reads occur when a transaction reads data elements that other concurrent transaction have updated. Figure 2.2 shows this phenomenon. $T1$ reads the element x, previously updated by transaction $T2$. Then, $T2$ makes further changes on x. The version read by $T1$ may be inconsistent because it is not the committed value of x produced by $T2$.

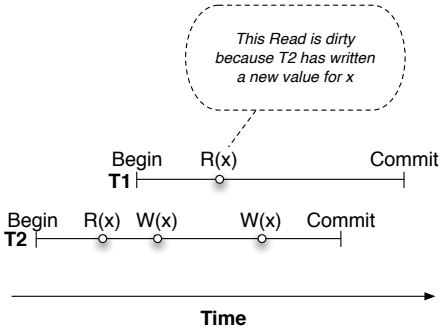


Figure 2.2: Dirty read problem

Non-repeatable read arises when a transaction reads the same data elements more than once and between those read operations, a concurrent transaction updates the same data elements. Figure 2.3 shows this scenario. $T1$ reads x twice, once before transaction $T2$ updates it and once after committed transaction $T2$ has updated it. Each read operation of $T1$ produce different values, which introduces consistency problems. The name of this problem arises because the original read is not repeatable.

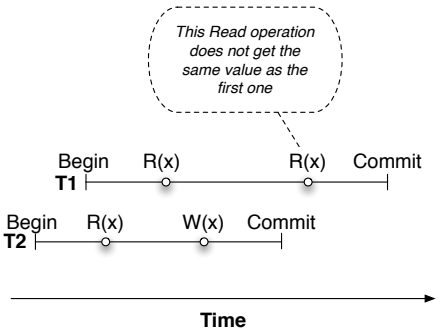


Figure 2.3: Non-repeatable read problem

Finally, phantom reads occur when some data elements appear and disappear inconsistently in the operations executed by concurrent transactions. The following scenario shows an example of this phenomenon. Transaction $T1$ reads a set of data elements satisfying some search condition. Then a concurrent transaction $T2$ creates new data elements that satisfy

$T1$'s search condition and commits. If $T1$ then repeats its read with the same search condition, it gets a set of data elements different from the first read.

In addition to these three different phenomena, another problem called *lost update* has been identified [GR93]. This problem occurs when two or more concurrent transactions read the same data elements and then update those elements based on the value originally read. Each transaction is unaware of the other transactions. When the transactions commit, the last of the updates overwrites the updates made by the other transactions, which results in lost data. The problem is depicted in Figure 2.4. The write operation on x made by transaction $T1$ is ignored by transaction $T2$, which writes the object based on the original value that it read.

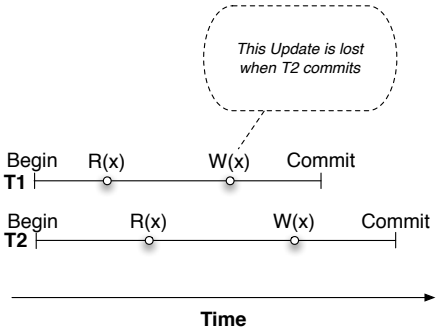


Figure 2.4: Lost update problem

ANSI isolation levels deal with these data consistency problems that may arise when executing concurrent transactions. The mechanisms used to solve data consistency problems are based on applying locking policies over the data that appear in queries. The following is a description of the four transaction isolation levels defined by ANSI:

- **Read-Uncommited** - There are no locks over the data elements read or updated by concurrent transactions. That is, a transaction can read data elements that have been updated by other transactions that are in progress (dirty read problem).

- **Read-Commited** - This is the default isolation level. Usually shared locks (incremental locks) are used while reading data elements. It ensures that a transaction will not read physically corrupt data elements that other concurrent transaction has changed and not has committed yet (this avoids the dirty read problem). However, it not ensures that the data elements will not be changed again before the end of the transaction (non-repeatable read problem).

- **Repeatable read** - Here, exclusive locks are placed on all data elements that appear in a query, so other concurrent transactions cannot update those elements. This avoids dirty read, non-repeatable read problems and lost updates[2] but it presents the phantom read problem.

[2]If two concurrent transactions update the same data elements using a single UPDATE statement and do not base the update on the previously retrieved values, lost updates cannot occur at the default isolation level of read committed.

- **Serializable** - It is the most restrictive isolation level. It also avoids the phantom read problem. It prevents other concurrent transactions from updating or inserting data elements into the data set of the current transaction until it is completed.

A critique of the ANSI isolation levels was done by Berenson et al. in 1995 [BBG+95]. Their study, analyzes the drawbacks of each ANSI isolation level and introduces a new isolation level called *snapshot isolation*. Snapshot isolation is a key concept in this Ph.D. Thesis and is described in the next section.

2.3.1. Snapshot Isolation

Snapshot isolation (SI) [BBG+95] is a multi-version concurrency control mechanism widely used in databases (e.g. Oracle, PostgreSQL, Microsoft SQL Server). SI provides a high level of isolation and avoids all ANSI SQL phenomena (lost updates, dirty reads, non-repeatable reads, phantom reads). One of the most important properties of SI is that readers and writers do not interfere. This is a big gain compared to serializability, the traditional correctness criteria for databases, where a locking implementation prevents concurrent reads and writes on the same element while optimistic concurrency control aborts the reader.

SI can be implemented as follows. The system maintains a counter C of committed transactions. At commit time, C is incremented and the new value is assigned to the committing transaction T as *commit timestamp CT(T)*. When a transaction T is started, it receives as *start timestamp ST(T)* the current value of C. When a transaction writes a data element x, it creates a new (private) version of x. When reading a data element x a transaction T either reads its own version (if it has already performed a write on x) or it reads the last committed version as of the start of T. That is, it reads the version created by a transaction T' so that $CT(T')$ is the maximum CT of all transactions that wrote x and $CT(T') \leq ST(T)$. By reading from a snapshot, reads and writes do not interfere. However, if two concurrent transactions want to write the same data element, SI requires one to abort. Such conflicts can be detected at commit time. When a transaction T wants to commit, a validation phase checks whether there was any concurrent transaction T' (i.e. $CT(T') > ST(T)$) that already committed and wrote a common data element. If such a transaction exists T aborts, otherwise it commits. If T commits, its changes (writeset) are made visible to other transactions that start after T commits.

Figure 2.5 shows an example with four transactions. It is assumed $C = 10$ and the transaction T with $CT(T) = 10$ updated x (not shown in the figure). Now $T1, T2$ and $T3$ start concurrently and all receive as start timestamp the value 10. $T2$ writes x. Its validation succeeds and $CT(T2)$ is set to 11. $T3$ reads the version of x created by T. Since it is read-only, no validation is necessary and it does not receive a commit timestamp. $T1$ reads the version of x created by T, and then writes x creating its own version. When $T1$ wants to commit, however, validation fails since there is a committed transaction $T2$, $CT(T2) > ST(T1)$, and $T2$ also wrote x. Therefore, $T1$ has to abort. Finally, $T4$ starts after $T2$ commits and receives $ST(T4) = 11$. It reads the version of x created by $T2$.

Databases do not implement this optimistic version of SI, where conflicts are checked at the end of the transaction. Instead, they use a pessimistic version based on write locks. When a transaction T updates a tuple it first obtains a write lock. If another transaction T' tries to modify the same tuple, it will be blocked until T finishes. If T commits, T' will be aborted.

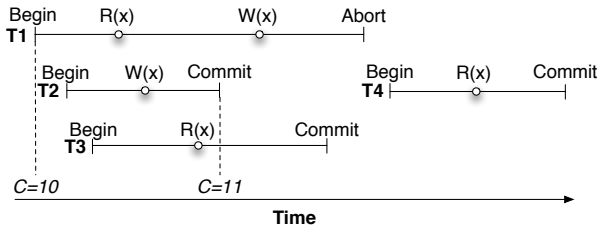


Figure 2.5: Snapshot Isolation

Otherwise T' will get the lock and can update the tuple. This pessimistic implementation detects conflicts early and therefore, avoids wasting resources in transactions that may abort.

Despite SI may increase the performance of transaction execution with regard to serializability, it introduces a problem related to consistency constraint violation. The following is an example of this problem, called *write-skew* [BBG+95]. Suppose C is a constraint between two data elements x and y. If $T1$ reads x and y, which are consistent with C, and then a $T2$ reads x and y, writes x, and commits. Then $T1$ writes y. If there were a constraint between x and y, it might be violated. Write skew could arise from a constraint at a bank, where account balances are allowed to go negative as long as the sum of commonly held balances remains non-negative. If $x = 100$ and $y = -50$ represent the account balances the consistency constraint is preserved $C = x + y = 100 + (-50) = 50 >= 0$. Then, both $T1$ and $T2$ read $x = 100$ and $y = -50$ creating their own snapshots. $T2$ writes $x = 75$ and commits because it preserves the consistency constraint in its snapshot: $C = x + y = 75 + (-50) = 25 >= 0$. If $T1$ writes $y = -80$ the consistency is also preserved in its snapshot ($C = x + y = 100 + (-80) = 20 >= 0$) and commits. However, the consistency constraint has been violated after $T1$ has committed: $C = x + y = 75 + (-80) = -5 < 0$.

2.3.2. Advanced Transaction Models

The traditional model of transactions, also called flat transactions, has been widely used in many conventional database applications such as airline reservation systems, car rental systems, banking... However, in 1981 J. Gray advanced several limitations of the transaction concept for certain complex applications [Gra81], such as object-oriented or distributed applications. Traditional transactions were assumed to last for short periods (few seconds or minutes). However, workflow-based applications may contain tasks that last for long periods (days or even months). Moreover, the structure of transactions is not flexible enough to provide high performance when used for transactional complex interactions with the user. For example, in travel agent systems, users may book a flight, a hotel and a car in the same transaction but using several interactions that may last for several minutes. In this context, keeping locks across invocations drastically reduces the concurrency in the system. It would be better to split the transaction into sub-transactions to reduce the context of the locks on data.

To overcome these limitations, several *advanced transaction models* were proposed by the database community. With regard to the traditional flat transaction model, these advanced transaction models provide new features, such as modularity, new failure handling mechanisms and intra-transaction parallelism.

One of the first advanced transaction models was *nested transactions* [Gra81, Mos81]. Nested transactions extend the notion that transactions are flat entities, allowing a transaction to contain any number of sub-transactions. These sub-transactions may contain, in turn, other sub-transactions, what forms a tree structure of (sub)transactions. The root of the tree is called *top-level* transaction, and keeps the ACID properties. Transactions that have sub-transactions are called *parents* and their sub-transactions are their *children*. A child transaction must start after its parent and terminate before it. If a sub-transaction commits, its effects are only visible to its parent. Moreover, the abort of one of its superiors will undo its effects. The effects of any sub-transaction become permanent (that is, visible to other concurrent nested transactions) only when the top-level transaction commits[3].

Another important advanced transaction model is SAGAS [GS87]. A SAGA is a long-lived transaction composed of several independent sub-transactions (steps), $T = S_1, S_2 \ldots, S_n$. The main contribution of SAGAS is the incorporation of *compensating transactions* [Gra81] to the transaction model. A compensating transaction C for a transaction T is a transaction that semantically undoes the effects of T, after T has been committed. In a SAGA, each sub-transaction S_i has associated a compensating sub-transaction C_i. Thus, to execute a SAGA the system must guarantee either the sequence: $T = S_1, S_2 \ldots, S_n$ or $T = S_1, S_2 \ldots, S_j, C_j \ldots, S_2, S_1$ for some $1 \leq j < n$. That is, either all the sub-transactions are completed or any partial execution is undone using the compensating sub-transactions.

During the following years, many other advanced transactions models appeared to satisfy the requirements of new or specific application domains [Elm92], e.g. Split Transactions, Flex Transactions, Polytransactions, S-Transactions...

In this thesis, advanced transaction models have been used to provide the required transactional properties (e.g. relaxed isolation, consistency...) that demand the current multi-tier and service-oriented applications.

2.3.3. Transactions in J(2)EE

In J(2)EE, transaction management is bound to the *Java Transaction Service* (JTS) [Sunc]. At the core of the JTS is the transaction manager, the software artifact that handles the execution of transactions in J(2)EE. The transaction manager provides an API called *Java Transaction API* (JTA) [Sun99] to demarcate the application functionality that is included in each transaction. Transaction demarcation can be done explicitly by the application developer either in the client or in the EJB source code (*programmatic transactions*/*bean-managed transactions* (BMTs)), or delegated to the container (*container-managed transactions* (CMTs)).

When using BMTs, the developers must demarcate explicitly the transaction in the source code using the operations provided by the JTA (*begin()*, *commit()*, *rollback()*...). BMTs make the source code of the application more complex, because the non-functional code of the JTA appears interwoven with the functional code of the application. CMTs contribute to ease transaction demarcation and maintain the source code clean. With CMTs, the container intercepts bean method invocations and demarcates transactions automatically when required using the JTA behind the scenes. This is possible because the EJBs methods are tagged with

[3]The nested transactions with this semantics in their nodes are also called *closed nested transactions*. In [Gra81], J. Gray writes about nested transactions whose effects are visible to the outside world prior to the commit of the top-level transaction. This kind of nested transactions are called *open nested transactions* (ONTs). ONTs are described in Section 2.3.4.1.

several transactional attributes to indicate the transactional behaviour to the container (e.g. whether a new transaction must be created or not before executing the method invocation, if a transaction must be already created before executing the invocation...).

2.3.4. Advanced Transactions in J(2)EE

The J2EE Activity Service (J2EEAS) [Sun06b] is an specification that allows to create advanced transaction models in the J(2)EE platform. The J2EEAS is based on the CORBA activity service [Objа]. J2EEAS provides the notion of an abstract unit of work, called activity, that allows to model a concrete task (transactional or not). With the support provided by J2EEAS, applications can create structures of activities to model complex business processes. Figure 2.6 shows an example. The dotted ellipses represent activity boundaries, whereas the solid ellipses are transaction boundaries. Activity $A1$ contains two nested transactions, while activities $A2$, $A4$ and $A5$ are not transactional. Activity $A3$ is more complex; it is a transactional activity, which in turn encompasses a nested transactional activity ($A3'$). Activities $A1$ and $A2$ are sequential, while $A3$ and $A4$ are executed in parallel.

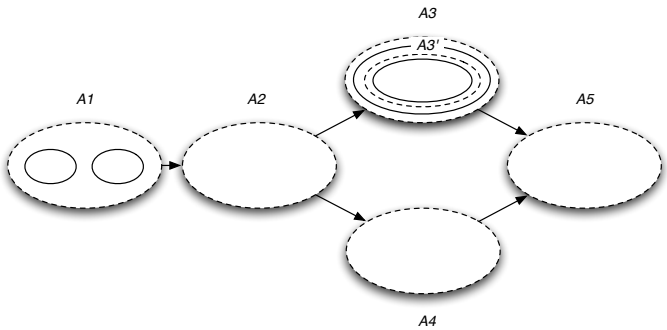


Figure 2.6: Activities and transactions

The activity service architecture consists of two components: the *activity service* itself and one or more *high level services* (HLSs). Figure 2.7 shows the architecture of the J2EE activity service.

The activity service is a generic state machine that manages *activities*, which may or may not be transactional. An activity may encapsulate a transaction or be encapsulated by a transaction. Moreover, activities may be nested within one another forming hierarchical structures. In order to demarcate activities, applications do not use directly the API of the activity service, but the API provided by a particular HLS.

HLSs use the functionality provided by the activity service in order to model the particular semantics of advanced transaction models. The developers of HLSs must provide the finite state machine, called *signalset*, that defines the state transitions for a particular transaction model. Depending on the current state, the activity service drives the concrete signalset to produce *signals* that trigger concrete *actions*. The execution of an action returns an *outcome* to the signalset that may influence a new state transition.

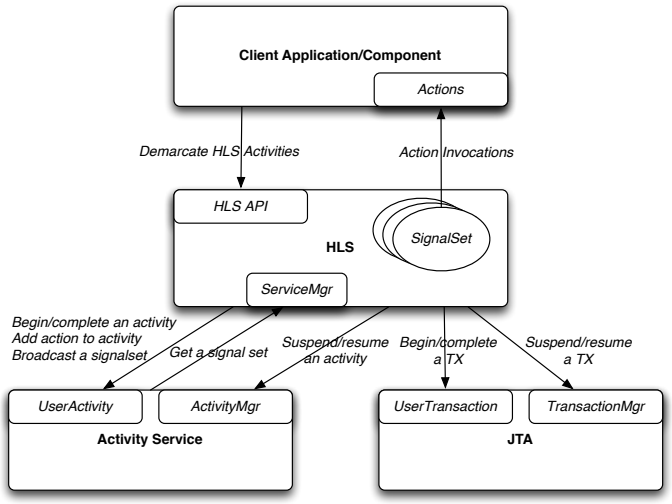


Figure 2.7: J2EE Activity Service architecture

2.3.4.1. Open Nested Transaction Model

The open nested transactions (ONTs) [Gra81, WS92, WV01] is a variation of the nested transaction model described in Section 2.3.2. The structure of an ONTs is the same as the structure of a nested transaction; a top-level ONT may contain any number of nested sub-transactions, which may recursively contain other nested sub-transactions organized in a tree structure. However, compared to a top-level nested transaction, a top-level ONT relax the isolation property. This means that, when a sub-transaction commits, its effects are visible outside the scope of its superiors in the tree (including the top-level ONT). Only sub-transactions preserve the ACID properties. Isolation is not kept at the top-level ONT because it would lock the resources used for long periods.

Chapter 3 uses an implementation of the ONT model adapted for the requirements of middleware environments based on objects/components. The concrete semantics is included the Appendix A of the *J(2)EE Activity Service* specification [Sun06b], and it is described below. The ONT model has been implemented as a high level service of the J2EE activity service and deployed as a service in a J(2)EE-compliant application server.

In the adaptation of the ONT model for middleware, each activity[4] represents an atomic unit of work (context) associated to a transaction. The creation of an activity implies the creation of the associated transaction. Activities may contain any number of nested sub-activities, which may again contain other nested sub-activities organized into a hierarchical tree. When a nested sub-activity is created, the transaction associated to its parent activity is suspended[5]. The parent transaction will be resumed, when the nested sub-activity finishes.

When a child sub-activity succeeds, the associated transaction is committed and all the resources used are released. Due to the relaxation of the isolation property, if the parent activity fails, atomicity must be guaranteed by logically undoing (compensating) the effects of its previously committed children sub-activities. Compensators are provided by the specific

[4]The model is described in terms of activities instead of transactions.

[5]This is done because many J(2)EE transactional service implementations do not support nested transactions.

applications and contain the necessary logic to undo the corresponding committed activities. Therefore, when a child sub-activity succeeds a compensator must also be registered with its parent. This procedure is performed recursively until the top-level activity is reached. When the top-level activity commits, the registered compensators are discarded.

When a sub-activity aborts, the transactions of all of its children that are still running are marked to rollback. For the already committed children, their respective compensators are applied in reverse order of completion.

2.3.5. Transactions in Web Services

One of the first specifications related to transactions in web services was the Business Transaction Protocol (BTP) [OAS02]. However, BTP was not supported by the industry and was deprecated. More recently, transactions have been addressed by WS-CAF [OAS05a] and WS-Coordination/WS-Transaction [OAS05b] specifications of the OASIS consortium[6]. Basically, all these specifications provide the means to define transactional contexts in applications and a set of coordination protocols (e.g. either two-phase commit (2PC) for ACID transactions or more sophisticated coordination for advanced transactions) managed by a coordinator entity.

In this Ph.D. Thesis, an early version of WS-CAF (0.1) was used to implement a prototype for evaluating the SOA replication framework, so this section is focused on its features.

The WS-CAF is a set of three related specifications: Web Services Context (WS-CTX), Web Services Coordination Framework (WS-CF) and Web Services Transactions (WS-TXM). As a whole, these specifications provide a stack of functionality for supporting applications or business processes that involve multiple web services requiring transactional support.

WS-CTX is the base specification on which the other WS-CAF specifications depend[7]. WS-CTX defines an extensible context structure and enables the different partners participating in a business interaction, to propagate, interpret and extend it. It provides an interface with the basic operations to manage the contexts: creation, update and termination of a context. The basic structure of a context contains a context identifier and can be extended to store additional information.

The fundamental idea underpinning WS-CF is the recognition of a shared and generic need for propagating context information in a web services environment, independently of the applications involved. The WS-CF specification defines a generic framework to coordinate the interactions among clients and the participant web services. As WS-CF is extensible, it allows different coordination protocols to be plugged-in. WS-CF complements WS-CTX by defining a coordinator element that is in charge of submitting notification messages to web services registered in a particular context. It provides an interface to register/unregister participant web services in the different contexts available in the WS-CTX. The coordinator is notified by the services of the WS-CTX when a particular context is created, updated or terminated in order to trigger the corresponding coordination actions.

[6] OASIS is a non-profit consortium that drives the development, convergence and adoption of open standards, including many specifications related to web services and SOA.

[7] As a consequence of merging the WS-TX and the WS-CAF, the WS-CTX specification was extracted from WS-CAF and standardized as an independent specification for defining generic contexts for web services.

WS-TXM defines three specific coordination protocols for modeling transactional interactions among web services using the WS-CF: ACID transactions, long running activities (LRAs) and business process transactions (BP). These transaction coordination protocols aim to agree on a common transactional outcome among the web services participating in transactions. ACID transactions is the basic transaction model. It defines a two-phase commit (2PC) coordination protocol among the participating services. The LRA model is designed for transactional business interactions that can last long periods and need to relax isolation. Within this model, an activity reflects a business interaction. Moreover, all the tasks performed within the scope of an application are required to be compensable. Therefore, the work done within an activity is either performed successfully or undone. Finally, BP transactions are used to model complex transactional interactions. This is useful when it is necessary to glue together disparate services and domains, some of which may not be using the same transaction implementation behind the service boundary.

Figure 2.8 shows two web services interacting using the WS-CAF. The requests and responses at the application level include the context structure defined through the services provided by the WS-CTX. The context structure includes information that allows the web services to behave properly. For example, when the service provider receives a request from the service consumer, the context includes information about who is the coordinator and what concrete protocol is being used in this interaction. In this way, if the service provider understands the protocol, it can be registered as a participant in the interaction. This is achieved by means of the control messages. The control messages are used to communicate actions between the application services and WS-CAF services. From this point on, the coordinator and the participant are able to exchange coordination messages of the specific protocol.

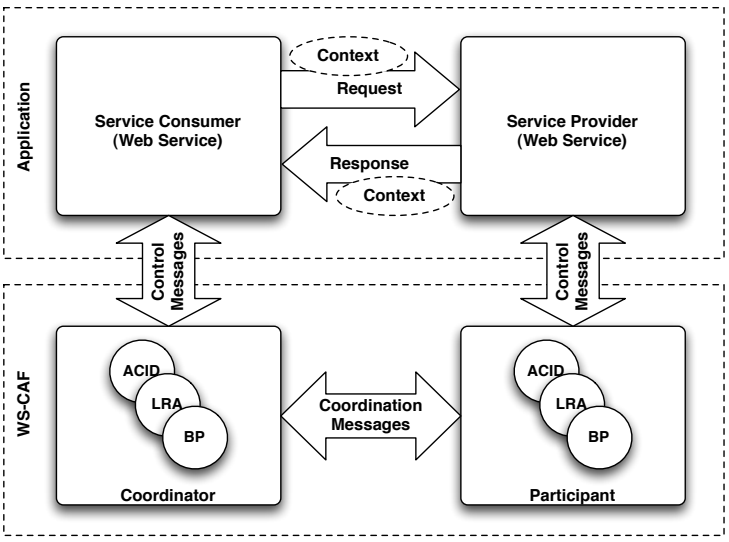


Figure 2.8: WS-CAF elements and interactions

2.3.5.1. Long-Running Activities in WS-CAF

Long running activities (LRAs) are used to model transactional complex interactions of web services that can last for long periods. LRAs relax the isolation property of transactions

in order to not to compromise shared resources for a long time.

In WS-CAF, the structure of LRAs is similar to the structure of open nested transactions. A LRA may have a set of ACID sub-activities that can be nested forming a tree of activities with parent/child relationships. The LRA model defines a *coordinator* element for the LRAs. The coordinator manages the correct the execution of each activity according to the LRA protocol. The LRA protocol consists of three messages: *Complete*, *Compensate* and *Forget*.

Complete is sent by the coordinator as a notification of successful completion of an activity. If a child sub-activity completes successfully, its effects are visible to other concurrent LRAs. As in open nested transactions, LRAs provide atomicity despite relaxing isolation using compensating activities. The LRA model defines an actor called a *compensator* that operates on behalf of a service to undo the actions it performs within the scope of an nested sub-activity. The coordinator of the LRA registers the compensators just before the context of each sub-activity terminates. The execution of a compensation is triggered upon the compensator receives the *Compensate* message sent by the coordinator. How compensation is carried out is obviously dependent upon the service (e.g. a cancellation action reverses a previous booking). As several sub-activities can be nested, the work performed by the sub-activities is required to remain compensable until the coordinator of the LRA informs, by means the *Forget* message, that the compensator is no longer needed. At that time, it can be garbage collected. If the compensation can not be done, an error message is thrown.

The following example presents an scenario where LRAs can be useful. Let us imagine a composite web service that allows its users to plan the perfect evening. This service involves other four different services performing reservations that must be carried out jointly: booking a taxi, reserving a table at a restaurant, reserving a seat at the theatre, and finally booking a room at a hotel. A user planning the perfect evening maybe needs about half an hour to complete properly all these reservations (e.g. getting information of concrete shows, hotels and restaurants, then checking their availability and finally performing different combinations with the different options available). If all of these reservations were executed within a single transaction, then the resources acquired during –for example– the book of the taxi, would not be released until the end of the transaction. This means that those resources will not be available to other clients trying to make their reservations concurrently. Here it is when LRAs come into scene. Instead of grouping the four reservations under a single transaction, each reservation can be modeled as an independent ACID sub-activity, nested in a top-level LRA that represents the whole work to be done. In this way, the work done in each sub-activity will not block unnecessary resources and other concurrent transactions using those resources could be executed.

2.4. Replication

In order to provide high availability to a system, it is required to introduce redundancy in it. This is typically done by replicating the functionality and critical state of the system in different locations. For example, when a software application requires high availability, a set of computers, or nodes, can be arranged as a *distributed system* or a *cluster*. Each node running an instance of the application is called a *replica*. A *replication protocol* is used to drive the coordination among the replicas. In this scenario, the replicas provide the necessary redundancy to the global system. If a failure occurs in one of the replicas, the system can tolerate the

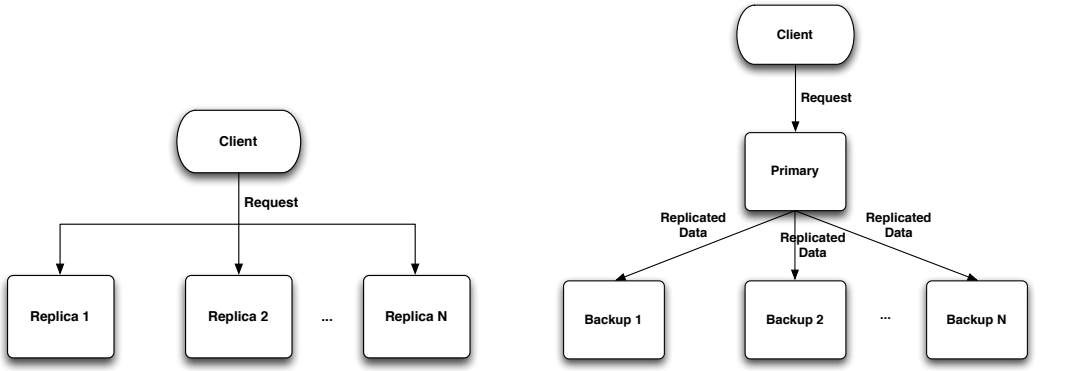
failure because other replica is able to continue performing the tasks. Applications are not only replicated for attaining high availability, but also to increase the system performance if the replication protocol is designed properly. With replication, higher performance is achieved by allowing concurrent accesses to the different replicas. It is necessary to point out that replication should be transparent for the clients. This means, that the clients of the replicated system should not be aware of the existence of replicas.

In the distributed systems area, there are two basic replication approaches: *passive* replication [BMST92] and *active* [Sch90]. Variations of passive and active replication can be found in [PBL91, VBH+91, DSS98].

An example of an active replication approach is shown in Figure 2.9 (a). In systems using active replication, all the client requests are processed in all the replicas. The replicas behave as state machines. Initially, each replica has the same internal state and its operations do not support non-deterministic code. Each client request must arrive to all the replicas in the same order. Then, each replica processes the requests in the received order. This means that the result obtained in each replica is deterministic because it depends on the initial state of the replica and the sequence of all deterministic requests already processed.

If some operations executed in a replica may be non-deterministic, it is possible to use *semi-active replication* [VBH+91, BMST92]. As it happens with active replication, in a semi-active replication approach, all the replicas process all the client requests. However, when a non-deterministic operation is going to be processed, the operation is redirected to a single replica appointed as *master*. After executing the non-deterministic operation, the master sends the result to the other replicas, named *followers*.

A passive replication approach –also called *primary-backup*– is shown in Figure 2.9 (b). In passive replication not all the replicas process all the client requests. One of the replicas, called *primary*, is in charge of receiving and processing all the requests from clients. Then, the primary collects the changes occurred in the state of the replica and sends them to the rest of replicas, that are called *backups*. Upon receiving the changes, the backups apply them locally.



(a) Active Replication (b) Passive Replication/Primary-Backup

Figure 2.9: Basic replication approaches

The major drawback of these approaches is that there is a loss of computing power in the set of replicas. In the active replication approach, all the replicas perform the same work. In passive replication, only the primary is performing computations. The backups simply wait

for the changes that the primary sends to them. In addition, passive replication requires a consensus mechanism to decide which replica acts as the primary. This mechanism is also triggered when the primary fails and one of the backups must replace it.

In the database area, different models and protocols for data replication have been studied and classified [GHOS96, WSP+00]. In [GHOS96], Gray et al. classify replication techniques in database systems with regard two parameters: *"where"* the transactions can perform updates on data, that is, in which replicas and *"when"* the updated data are propagated among the replicas.

From the point of view of "where" the data can be modified, there are two possibilities: the centralized approach –equivalent to the primary-backup described above– and the distributed approach, called *update-everywhere*. As its name points out, in the update-everywhere approach any replica can receive request to modify data. When the request is received by a replica, that replica behaves as a primary replica in the primary-backup scheme. The main challenge is to guarantee the data consistency in all the replicas. The update-everywhere approach is shown in Figure 2.10.

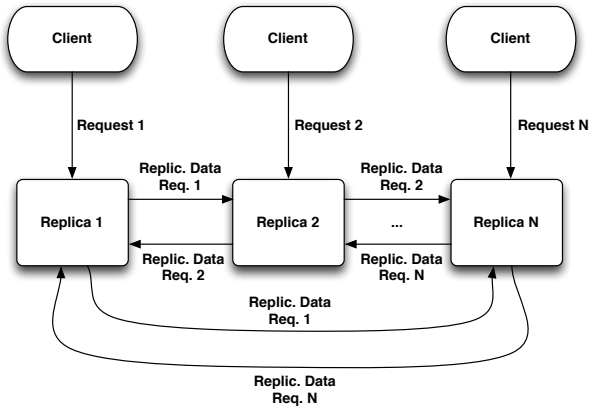


Figure 2.10: Update-everywhere approach

From the point of view of "when" to replicate changes in data replication, there are also two options: *eager* and *lazy* replication. In eager replication, changes are propagated to all the replicas before the end of transaction where changes occurred. On the other hand, in lazy replication changes are propagated to the replicas once the transaction has finished. In [GHOS96], J. Gray et al. proved that eager replication is not scalable using distributed locking. Distributed locking causes many deadlocks and transactions can not progress. This means that if new replicas are added to the cluster, the global performance is not increased. This paper was the basis for the study of different replication protocols based on eager replication with scalable features [KA00b, KA00a, LKPJ05]. These protocols do not process symmetrically the transactions in the replicas, but asymmetrically. In an asymmetric protocol, each transaction is executed locally in one replica and the changes are replicated and applied directly in the other remote replicas. In order to keep the replicas consistent, it is more efficient in terms of processing time to apply the remote changes received directly in the local database than execute locally a new transaction on the local data as symmetric protocols do.

2.4.1. Replication in Multi-tier Architectures

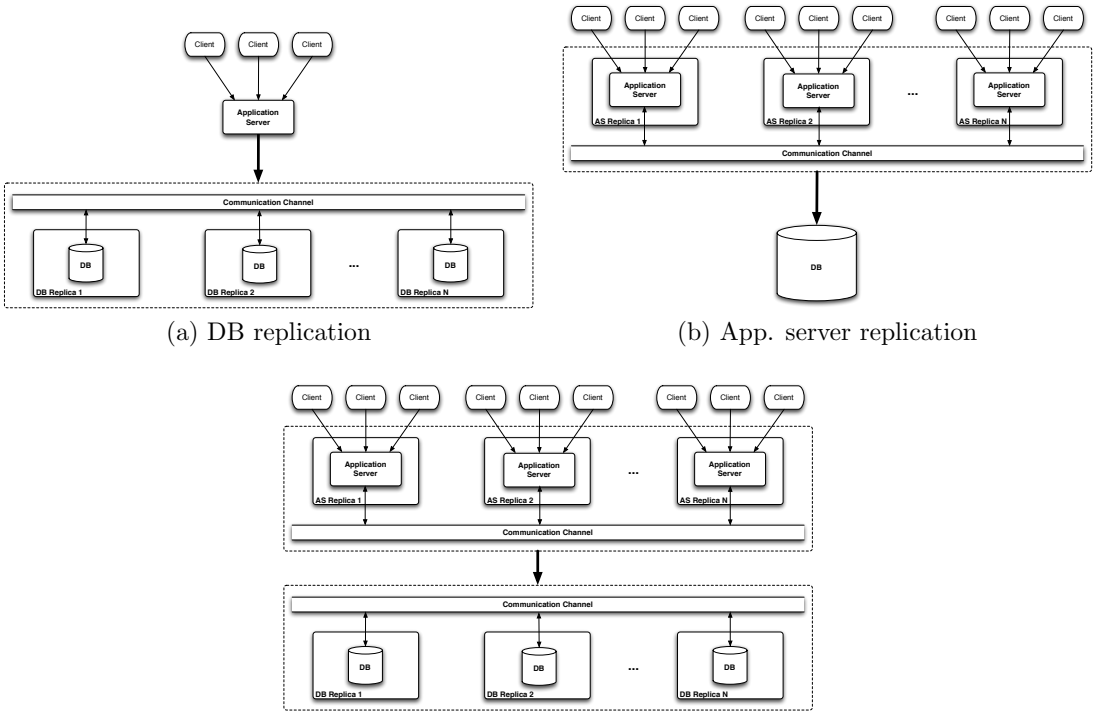
In the last few years, a lot of work related to replication in multi-tier architectures has been done [NMM02a, ZMM02, BLW02, BBM+04, LR04, WKM04, ZMM05, WK05, SPH06]. The great success of middleware platforms in the industry such as CORBA, .NET or J(2)EE has to do with this fact. The companies have more and more applications deployed in these environments, some of them critical for their businesses. However, many application servers that support these applications lack of the required support for high availability and scalability that current applications are demanding. Now this trend is changing and replication techniques are being introduced in application servers to try to provide these two features using clusters.

Multi-tier applications hold state and data that must be kept consistent along the tiers. The persistent data used by applications are stored in the back-end tier, that is usually a database. The application state is composed of in-memory components that constitute the business logic of the application. This state can be divided into session or persistent state. Session state is usually stored in the presentation or in the middle tier. In this thesis we consider that session state is hold in the middle tier. Session state is related to particular clients, so each client has its session state hold in its own components. Finally, persistent state contains cached data in the middle tier fetched from the back-end tier. The persistent state is hold in components that can be accessed concurrently by the clients. Thus, when replicating a multi-tier architecture both, the application state and data hold in the middle tier and the back-end tier must be taken into account. Moreover, the consistency of the application must be preserved in all the replicas.

As there are no concurrent accesses to the session components, it is relatively easy to replicate them. However, the replication of persistent components and data in multi-tier applications requires a great effort regarding to maintain consistency in the cluster [KJPS05]. In a multi-tier architecture another parameter to classify replication can be introduced: *"how"* the replication can be achieved through the architecture. In a multi-tier architecture, each tier can be replicated independently, what is called horizontal replication. Figure 2.11 shows several replication configurations following this approach. The two figures at the top show the replication of only one tier: (a) the back-end tier, represented by a database, or (b) the middle tier, represented by an application server. In both cases, only one replication protocol is required. The major drawback of these two configurations is that the non-replicated tier (either the database or the application server) becomes a potential single point of failure for the cluster. The figure (c) at the bottom, shows the replication of both, the middle tier and the back-end tier independently. This configuration avoids single points of failures. However, it is complex to implement and manage because it requires two replication protocols (one per tier) and additional control to coordinate both protocols.

Figure 2.12 presents an alternative to horizontal replication solutions called *vertical replication*. In this approach, each replica consists of an application server instance connected to an instance of the database. As the replication process only takes place at the application server level, only one replication protocol is required. Therefore, each application server is responsible to synchronize properly their state with the data stored in their local database as it happens in a non-replicated system. With this approach, neither the middle-tier nor the back-end tier are single points of failure for the cluster.

The vertical replication approach is the architectural solution used by the replication protocols for multi-tier architectures presented in this thesis. The protocols for high availability



(a) DB replication

(b) App. server replication

(c) App. server and DB replication

Figure 2.11: Horizontal replication configurations

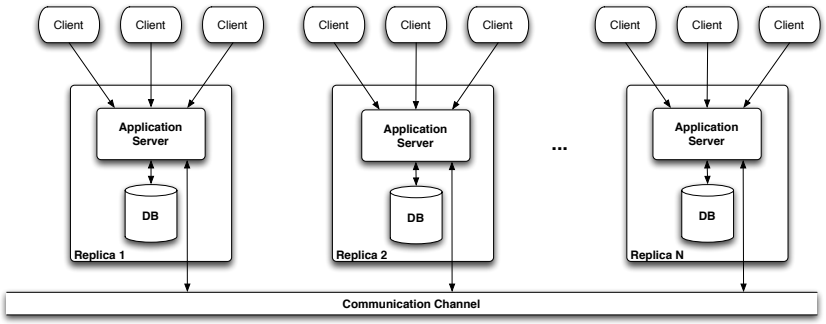


Figure 2.12: Vertical replication approach

combine vertical replication with a primary-backup approach. However, the protocol that provides also scalability combines vertical replication with an update-everywhere approach. In this case, the replication protocol must guarantee that all the replicas commit the same transactions despite the concurrent execution of different client requests at different replicas [KJPS05].

2.4.2. Replication in Service-Oriented Architectures

Some of the services that participate in a service-oriented application may also be critical for organizations. In order to provide high availability to these services, several replicas of each

critical service may be deployed in distributed nodes, either connected through LANs or WANs. So, the replication techniques used in SOAs are similar to the ones used in other distributed systems (e.g. primary-backup or active replication schemes may be used).

However, when replicating web services it is important to take into account two requirements: (1) they use the SOAP protocol to interact in a standard manner one with each other and (2) they may distributed in WAN environments. Thus, the solutions that aim to provide high availability for web services must preserve these two requirements with the help of the underlying infrastructures in which they are deployed (e.g. GCS to guarantee reliable communication and group membership, web service engines to provide SOAP communication...). Current solutions for replicating web services [BvRV04, SLK04] do not preserve completely the requirements of web service-based infrastructures.

The replication protocol for service-oriented architectures used in this thesis is based on the active replication schema, preserves the standard access to the service and its autonomy and allows replication in nodes connected through WANs.

2.5. Recovery

Unexpected failures (e.g. a power outage) may cause the crash of a computer system. After the failure, if the system is not seriously damaged, it can be restarted. Apart from the loss of availability, a crash failure is not a problem for the consistency of stateless applications and services. However, the failure can cause inconsistencies in stateful applications and services that were executing client requests in the system before the crash. When a system crashes, the data stored in volatile memory is lost, but the data stored in persistent storage remains intact. If the system is fault-tolerant, a *crash recovery process* is launched during the restart phase to bring back the most recent and consistent state to the system [WV01].

The recovery process can be done both, offline and online [LK08, Jim09]. In *offline recovery*, the system does not process new client requests whilst the recovery process is being performed. Of course, offline recovery breaks the availability of the system with regard to its users. On the contrary, *online recovery* allows to continue processing client requests during the recovery phase, and thus, it preserves the high-availability of the system.

In fault-tolerant clusters providing high availability through replication and requiring high throughput guarantees, online recovery is the only option to be considered. The failure of an individual replica must not cause the failure of the whole system. After the failure of a replica, the recovery process may be performed on the failed replica, called the *recovering replica*. The recovery process will bring the recovering replica back to the cluster with a consistent state synchronized with the other replicas of the cluster, called *recoverer replicas*. Online recovery can be also used to scale out a high performance cluster. In a cluster with the proper replication protocol, adding new replicas can increase the number of client requests processed and thus the performance of the system as a whole.

In multi-tier architectures with vertical replication, the recovering replica can recover the middle tier and the back-end tier at the same time using the state information received at the application server level from the recoverer replicas. In this case, the recovering replica must guarantee that its local database is also updated, so the state received at the application server is persisted in the local database. A challenge arises when a replica needs to recover both tiers separately. This can occur when the state of the recovering replica can not be recovered

with the current in-memory information hold in the recoverer replicas. In this scenario, a snapshot of the database must be sent to the recovering replica. Upon receiving and installing the database, the recovering replica completes its state synchronization using the information received at the application server level from the recoverer replicas.

2.6. Middleware Caches

Caches are present in almost every hardware and software architecture. A cache can be defined as a high-speed access intermediate store that manages recently and/or frequently accessed data from a data source [CDK01]. Caches are collocated closer to the clients than the original data sources. When a client process requests data, the contents of the cache are checked first. If up-to-date copies of the data are found, the data are supplied from the cache. Otherwise, up-to-date copies of the data are fetched in the cache from the data source, replacing some existing data if necessary. Thus, a cache avoids when possible to access data that are frequently accessed on the original (and possibly slower) data source. Caches play an important role because they can improve the performance of a system and avoid or alleviate bottlenecks in shared resources.

In the application server internals, there are multiple types of caches. For example, the pool of threads that serve client requests or the pool of database connections for an application can be considered as caches. In this thesis we are interested in the so-called *second-level caches*. This kind of caches are currently used in middleware systems in order to increase the performance of the applications that access persistent data. In an application server, a second-level cache is a set of in-memory data elements fetched from a persistent data source. These data elements are synchronized with the original data source and can be accessed by all the client sessions. This allows to minimize the interaction with the underlying data source when the data elements are accessed several times. However, this also means that the state of the data elements must be kept consistent when they are accessed concurrently. Moreover, these caches must provide good performance for both, intensive read-only and read-write operations. Without a second-level cache, every time a particular data element is referenced in a transactional context, the application server must refresh or create a new instance accessing the data stored in the underlying data source.

Figure 2.13 shows how second-level caches are integrated in the architecture of an application server. In the figure, the clients are are accessing two application components A and B through their corresponding client sessions. The components A and B are the in-memory representation of data stored in a persistent storage, in this case a database. When each component is accessed for the first time, the object/relational mapper (O/R mapper) retrieves the data from the persistent storage. Then, the O/R mapper –instead of creating a component instance of the data for each client session– creates a unique instance of the component and puts it in the second-level cache (shown in grey in the figure). Finally, a reference to the instance is returned to the corresponding client sessions. This avoids unnecessary accesses to the persistent storage in further accesses to the components A and B from client sessions.

In high performance clusters following an update-everywhere approach, the state of the second-level caches must be kept consistent in all the replicas because the data elements that they contain can be accessed and modified concurrently not only in the same replica, but also in different replicas [WV01]. Moreover, in a vertical replication approach, when the applications

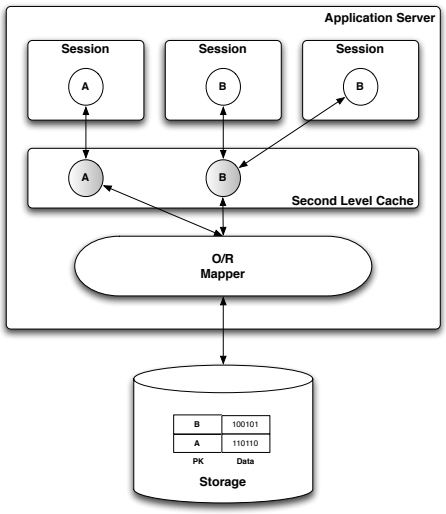


Figure 2.13: Second-level cache in application servers

modify the state of an data element in the cache, the cache and the underlying storage must be kept synchronized to guarantee the consistency of the replica.

2.7. Group Communication Systems

The communication model used in the protocols described in this Ph.D. is based on the features provided by the so-called *group communication systems* (GCSs) [CKV01]. A group consists in a set of nodes called members. Groups are used to manage data in replicated systems where different members cooperate to achieve a common objective by receiving and processing the same set of messages [CDK01]. In the context of the thesis, the group members are the processes/threads in each node of a distributed system that manage communication with the other nodes.

GCS provide two main services: group membership service (GMS) and multicast communication [Bir97].

The GMS is responsible of providing the notion of *view*. A view contains the current connected and active members of a group (membership). The membership of a group can be established statically before starting the nodes. However, this approach is less flexible than allowing the members of the group to join and leave the group at any moment of the system's lifetime. Practical GCSs implement this feature –called dynamic membership– in their GMSs. The main tasks of a group membership service are [CDK01]:

- Provide an interface to manage groups. That is, to create/destroy groups, add/remove members to a particular group...
- Notify members of changes in the composition of a view. Changes in the composition of a view (*member crashes*, *addition of new members to the group*, *connections lost with some members*...) are eventually delivered to the application.

- Implement a failure detector for being aware of replica failures. This is done by monitoring the group members. To detect failures in members is important because it can change the composition of the view. The decision to exclude a member of a group is usually taken using a consensus algorithm.

- Perform group address expansion. Usually, members send messages using the group identifier instead of the list of the members belonging to that group. This facility provided by the GMS performs automatically the expansion of the group identifier into the current group membership. This way the GMS can decide where to deliver a message even if the membership changes during delivery.

Without a multicast service, when a member sends a message to a group of N servers, it must send the same individual message $N-1$ times between itself and each other member of the group. This is not the best communication model for the scenarios presented in this thesis. Multicast communication enables to send a message to a group through a single operation, avoiding the flooding of the network with redundant messages. Multicast communication is usually implemented using IP multicast [CDK01]. The major drawback of IP multicast is that it does not provide ordering or reliability guarantees [HT93]. GCSs alleviate these drawbacks, allowing specify different reliability and ordering guarantees for multicast communication.

Reliable multicast ensures that all available members deliver the same messages. *Uniform reliable multicast* extends the previous definition to ensure that if a message that is delivered by a member (even if it fails), it will be delivered at all available members. Reliable multicast requires the following properties [HT93]:

- **Integrity** - A member delivers a message m at most once.

- **Validity** - If a member multicasts message m then it will eventually deliver message m.

- **Agreement** - All the members in the group will eventually deliver a message m if any of them has delivered it. This condition is related to atomicity applied to delivery of messages to a group.

With regard to the ordering guarantees of the messages in multicast communication, these are the options [CKV01]:

- **FIFO ordering** - If a member issues a message m before another message m', then if the receiver delivers m', m will be delivered before m'. That is, the messages of the same sender are delivered in the same order in all the processes of the group.

- **Causal ordering** - It extends FIFO order requiring that a response m' to a message m is always delivered after the delivery of m.

- **Total ordering** - Extends the causal order requiring that if a member delivers m before m', then any other member that delivers m' in the group delivers m before it.

Group membership and multicast services are strongly interrelated. Changes in the composition of a view (e.g. *member crashes* or *addition of new members to the group*) are eventually delivered to the group members using multicast for communication purposes.

Two properties specially desired in replicated systems are *primary component membership* and *virtual synchrony* [CKV01]. In primary component membership, views installed by all members are totally ordered (there are no concurrent views), and for every pair of consecutive views –V_i and V_{i+1}–, there is a process that survives from the view V_i to the second V_{i+1} (e.g., does not crash between the installations of these two views). Virtual synchrony ensures that two members transiting from view V_i to a new view V_{i+1} have delivered the same set of messages in view V_i. This property is required to build fault-tolerant systems and eases the task of transferring state to new members of a group. *Strong virtual synchrony* [FvR95] reinforces virtual synchrony ensuring that multicasted messages are delivered in the same view that they were sent (sending view delivery).

Several group communication toolkits have been developed in the last two decades. Isis [Cor] was the first one, but many others have appeared since then: Transis [ADKM92], NewTop [EMS95], Totem [MMA+96], Horus [vRBM96], Ensemble [Hay98], JGroups [JGr] and Spread [Spr, ADR00]...

The facilities provided by GCSs (group membership, reliable multicast...) have been used to implement the protocols proposed in this thesis.

CHAPTER 3

High Availability in Multi-Tier Architectures

Experience is what you get when you didn't get what you wanted.

– Anonymous (*Italian proverb*)

Nowadays, many enterprise business applications are typically built and deployed on top of middleware platforms based on multi-tier architectures. These applications require high availability support to prevent financial losses and/or service level agreements violations due to unreliable services or system crashes. Moreover, they include transactional interactions that are getting more and more complex. In such applications, traditional transactions may become a bottleneck in performance because they tend to lock the resources for a long time. Long running activities fit better to this kind of interactions.

This chapter presents a set of replication protocols that support highly available transactions and long running activities in multi-tier applications deployed in clusters.

3.1. Introduction

The enterprises are demanding high available solutions for their multi-tier applications. Replication is used to attain high availability in many middleware products. However, current solutions usually fail to satisfy the industry expectations due mainly to two reasons:

- **They do not provide highly available transactions** - In the advent of a replica failure, current solutions abort all ongoing transactions, and the application is forced to re-execute them, what results in a loss of availability and transparency. However, in some applications, e.g. workflow systems or orchestrated web services, there is no way to

replay the aborted transactions. The same problem exists for long running activities, mostly ignored by current middleware platforms.

- **They are focused on the replication of a single tier** - Recent work on multi-tier replication, only replicates the application server (middle tier) while using a single shared database instance among the application servers [FG01, FG02, LR03, WKM04, WK05]. These approaches are called *horizontal replication* because they replicate the data across a single tier. The main shortcoming is that the shared database may become a bottleneck and a single point of failure. When the non-replicated tier crashes, e.g. the database, the global availability of the system is lost and the applications cannot process client requests. An alternative is to replicate both tiers independently, but the solutions require complex interactions between the replication protocols of each tier.

This chapter presents a novel *vertical replication* support for both, the application server and the database tiers, providing highly available transactions and long running activities. A set of replication protocols based on vertical replication and the primary-backup approach are introduced. The protocols guarantee exactly-once semantics for client requests [FG01, FG02] and are able to deal with several client interaction patterns (1 request/1 transaction, N requests/1 transaction, 1 request/N transactions and N requests/M transactions). Moreover, the protocols provide transparent failover to the clients of the applications running on top of the replication platform. With the features provided by these protocols, it is possible to build high availability clusters for stateful applications.

In this context, the provision of high availability for transactions and transparent failover raises other challenges. For example, unlike previous work, the solution has to coordinate the work done by the container of application components with other services provided by the application server such as the transaction manager, the database connection manager, the activity service engine...

The replication protocols presented here have been implemented and evaluated in a multi-tier architecture based on J(2)EE. JBoss [Reda] has been the target application server where the protocols have been integrated. In order to test the solution with long running activities, a prototype of the J(2)EE Activity Service [Sun06b] and the open-nested transaction (ONT) model have been also implemented. The performance of a transactional application has been evaluated with the ECPerf benchmark [Sun03a]. In order to evaluate the long running activities, a custom benchmark has been used.

The remainder of the chapter is structured as follows; Section 3.2 refreshes some background concepts. Then, Section 3.3 shows the system model used. Section 3.4 presents the replication protocols. The evaluation of the protocols is shown in Section 3.5. Finally, Section 3.6 concludes the chapter.

3.2. Background

The replication protocols presented in this chapter provide high availability to transactional J(2)EE multi-tier applications deployed in clusters.

JBoss [Reda], a well-known open-source J(2)EE application server, and the ADAPT replication framework [BBM+04] have been used as a base for implementing and deploying the

replication protocols. Figure 3.1 shows an schema of the main J(2)EE application components and services that are present in a replica. The figure shows how these elements are integrated.

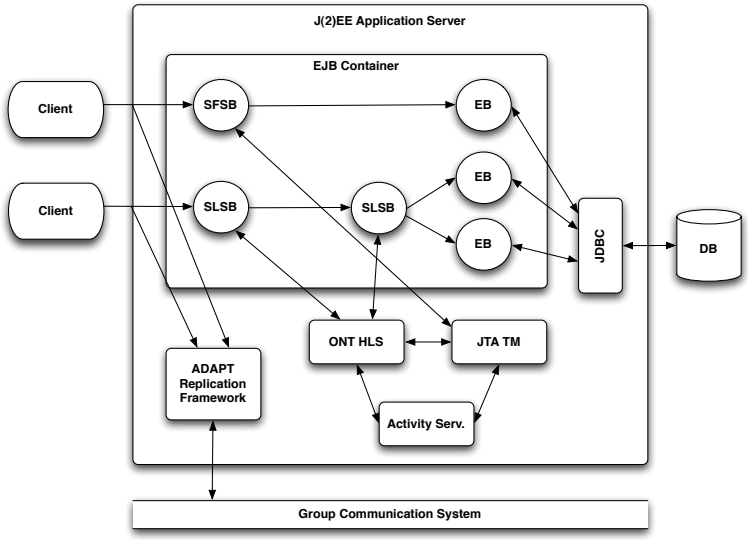


Figure 3.1: Main J(2)EE components and services and their relationships

J(2)EE applications are usually composed by multiple EJB components managed by an *EJB container*. Application clients –either thin (e.g. web components) or fat (e.g. Java applications)– invoke the required methods on the front-end EJB components to carry out specific tasks. The EJBs may call other EJBs to fulfill their functionality. The EJB components may also create transactions and activities using the services provided by the J(2)EE application server. In J(2)EE these services are respectively the *transaction manager* (TM) –which offers the Java Transaction API (JTA) in order to demarcate transactions– and the *activity service*. As the activity service is a generic service, the EJBs must use specific APIs called *high level services* (HLSs) to demarcate concrete activities representing advanced transactions. In the figure, a specific HLS is shown (ONT HLS). The ONT HLS allows the EJBs to demarcate open nested transactions. Internally, the ONT HLS demarcates the transactional activities using the JTA TM. Additionally, when the EJBs components need to save their state in persistent storage, they use the *Java Database Connectivity service* (JDBC) to save the state into databases (DBs).

The *ADAPT replication framework* [ADA] has been used to ease the implementation of the different replication protocols. This framework was designed to be plugged into the JBoss application server and supports the prototyping of different replication protocols, hiding all the underlying complexity of the application server internals. This is achieved by means of the use of interceptors in the request/response invocation chains and through an API that simplifies the management of the EJB components to the developers of replication protocols. The ADAPT framework provides two types of interceptors:

- **Client component monitor (CCM)** - The CCM intercepts all the outgoing invocations at the client side. It is used to implement the client side of the replication protocols (e.g. it allows to attach additional information to the request –such as client and transaction

identifiers– and allows to resubmit client requests to the new primary when a failure is detected in the current one). The CCM is dynamically loaded by the client when getting the stub of a remote EJB. There is one single CCM instance per client.

- **Component monitor (CM)** - The CM is the server side counterpart of the CCM and is responsible for intercepting both, remote invocations from the client to the EJBs, and local invocations between EJBs. In the CM is implemented the server side of the replication protocols.

With the vertical replication approach, replication is not made along tiers (e.g. replicating the application server independently of the database), but across them (e.g. replicating pairs of application server and database). The independent replication of tiers is far from being trivial, and requires some sophisticated logic to enable the consistent integration of both replicated tiers [KJPS05]. The combined replication of tiers enables an integral replication solution fully achieved at the application server and without modifying the database internals. This fact is important for pragmatic reasons since it enables the use of the replication protocols with any existing DBMS and without requiring access to the source code.

In order to provide transaction-aware availability to J(2)EE multi-tier applications, it is necessary to collect the changes in stateful components that are produced in each transaction interaction. In a primary-backup approach, these changes are collected in a *writeset* by the primary replica and sent to the backup replicas of the cluster at the end of each transactional client interaction with the system. Message driven beans are stateless components and thus, they do not need to be considered. Stateless session beans (SLSBs) only last for a single request and therefore they do not need to be considered either. Stateful session beans (SFSBs) can keep state across several requests of the same client so they should be checkpointed to provide high available transactions. Finally, entity beans (EBs) store persistent data, so they should be checkpointed as well.

In applications, SLSBs and SFSBs can invoke each other within the same interaction (e.g. a SFSB may call a SLSB or another SFSB) and this may affect the length of their lives. This fact needs to be considered when creating the checkpoint of components. Only those SFSBs for which there is a path from the client going exclusively through SFSBs, maintain their state across client invocations, and only them need to be checkpointed.

Transactions due to their stateful nature are not reproducible. That is, once a transaction has been committed, there is no way to find out which results it produced. This may raise a problem during failover in a cluster when the primary replica fails. This problem arises when the former primary, at the end of a transactional interaction, sent the writeset to the backups, committed the transaction locally but did not sent the response to the client. During the failover, the client sends the request to the new primary. The new primary –that already applied the changes– receives the client request executes it as a new one. The re-execution of the client request may cause inconsistencies in the database. The solution is to checkpoint client response together with the updated data. Upon failover, the checkpointed response of the committed transaction can be returned to the client from the new primary without re-executing the request.

Long-lived transactions and ONT activities may span several client interactions. When each interaction finishes, there might be updates that have been committed and updates that are still uncommitted. In order to provide high availability to these transactions and activities, when propagating updates from the primary to the backups at the end of each interaction, both

committed and uncommitted changes must be checkpointed. Committed changes should be checkpointed to enforce their durability and exactly once semantics and uncommitted changes should be checkpointed to continue processing the transaction when a failure occurs.

The replication protocols use the *group membership* and *multicast communication* facilities of group communication systems (GCSs) to ease certain tasks. Group membership provide the notion of view (currently connected and active group members). The changes in the composition of a view (*member crash* or *new members*) are eventually delivered to the application. Here, it is assumed a primary component membership. In a primary component membership, views installed by all members are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one member that survives from the one view to the next one. The reliable multicast communication ensures that all available members deliver the same messages. *Uniform reliable multicast* ensures that a message that is delivered by a member (even if it fails), will be delivered at all available members. The *FIFO order* guarantees that all messages sent by a group member are delivered in first-in first-out (FIFO) order. Moreover, it is assumed that multicast messages are delivered also to the sender of the message (self-delivery). Finally, strong virtual synchrony ensures that messages are delivered in the same view they were sent (sending view delivery) and that two members transiting to a new view have delivered the same set of messages in the previous view (virtual synchrony).

By combining these concepts and technologies (primary-backup, vertical replication, transactions, GCS...), it is possible to build transaction-aware and high available clusters for multi-tier applications.

3.3. System Model

The replication approach is based on the primary-backup and the vertical replication schemas, both presented in Chapter 2. The replication model is depicted in Figure 3.2.

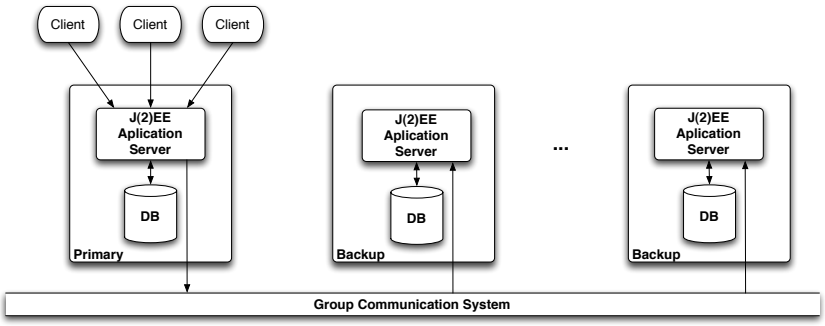


Figure 3.2: Replication model

In the primary-backup approach, one of the replicas, called primary, receives all the client requests. After processing the requests and before returning the response to the client, the changes are sent to the backup replicas. In the vertical replication schema, a *replica* is considered as the pair J(2)EE application server and database management system (DBMS). The set of all these replicas conform a *high availability cluster*.

In the system model only crash failures of the replicas are considered. A replica is considered crashed if the application server, the database or both fail. If the cluster contains N replicas ($N > 1$), it supports $N - 1$ failures. If the primary fails, a backup will take over and become the new primary. From that point on, clients will interact with the new primary. Moreover, the failure of the primary replica is transparently masked to the clients. Upon failover, the processing continues at the point that the previous primary was when it failed.

In the cluster, the primary replica sends the changes performed on the stateful EJB components to the backups using the facilities provided by a group communication system (GCS)[1]. The GCS running in each application server requires strong virtual synchrony and must be configured to use primary component membership. The messages are delivered using uniform reliable multicast with FIFO order guarantees.

3.4. High Available Replication Protocols

In this section the suite of replication protocols for high available transactions is presented. First is presented a replication protocol for transactions whose lifetime is a single client invocation to simplify the presentation (1 request/1 transaction interaction pattern). The first protocol is extended to support client-demarcated transactions where the clients may invoke several times the application components in the application server within a single transaction (N requests/1 transaction interaction pattern). Finally, the last protocol presented, deals with replication of long running activities based on ONTs (1 request/N transactions and N requests/M transactions interaction patterns).

The replication protocols provide the following consistency properties in the absence of catastrophic failures (all replicas fail):

- **Exactly once execution** - Every request is processed exactly once despite replica failures. If a transaction commits, the transaction is committed exactly once at each running replica. Every activity and every compensator are also executed exactly once.
- **Replica state consistency** - After a client receives a reply, it is guaranteed that all running replicas have the same state. That is, the backups contain the same set of checkpoints and if the failover is performed, the backups will reach to the same state that the primary had (the same SFSBs with the same state, the same committed EB updates, the same database committed state, the same uncompleted transactions and uncompleted activities with the same associated updates and reads, and the same activity trees).
- **Highly available processing** - Every client request eventually receives its outcome despite replica failures. That is, from the client perspective, transactions and activities never abort due to replica failures.

It should be highlighted that the first and second properties are provided by previous work (e.g. in [FG02] and [WKM04]), but only in the context of a single replicated tier (the application server). The protocols proposed here fulfill these two properties replicating both

[1]Note that GCSs are not required by any replication technique, but they largely simplify the implementation.

tiers and therefore, without exhibiting any single point of failure. Moreover, to the best of our knowledge, this approach is the only one that fulfills the third condition.

The next sub-sections detail each one of the replication protocols. The protocols describe the request and responses intercepted by the ADAPT framework from three different points of view: the client, the primary replica and the backup replica.

3.4.1. One Request Transactions

This replication protocol only considers transactions that start and complete during a unique client invocation (1 request/1 transaction pattern). During the invocation of a transactional method, the EJB may invoke other methods on other EJBs to accomplish the business logic. The changes made on these invocations are part of the same transaction. That is EJB methods that demarcate their own transactions are not considered by this protocol. The replication process is done when the transaction has been completed (lazy replication).

The protocol can be summarized as follows; For each client invocation to a transactional method of a front-end EJB, the EJB container of the primary replica creates a transaction. Then, the stateful EJBs (SFSBs and EBs) that have changed their state along the invocation are collected in a writeset. Then, the transaction is committed and the writeset is multicast to all the replicas together with the response. Finally, the response is sent to the client. The backups receive the writesets in FIFO order and apply the changes in the EJB container. Then, the backups persist these changes in the database.

3.4.1.1. Protocol Details

The pseudocode for the client part of the protocol is shown in Figure 3.3. The client, as part of the protocol, associates to each method invocation a unique request identifier (*r_id*). A r_id consists of a client identifier (*c_id*) that identifies univocally each client, and a request number (*r_no*) that identifies each concrete invocation of each client. This is done transparently by the client stub (lines 1-4), so it is not necessary to change the client application code. The r_no is increased every time a client receives an incoming response from the EJB container (line 6).

At the server side, the EJB container of the primary replica is in charge to control and store the changes made on stateful EJBs by the different transactional client requests. The replicas also store the responses sent back to the clients in order to deal with possible crash failures of the primary (See next section). These data are stored in two tables, both indexed by the r_id element. The changes are stored in the writeset table (*ws_t*) and the client responses in the last response table (*last_resp_t*). The pseudocode of the protocol for the primary replica is also shown in Figure 3.3.

Assuming that no failures have occurred, when the primary replica receives an EJB invocation from a client, it starts a transaction (line 11). Then, the business logic of the method is executed for that request in the recently created transaction. The business logic of the EJB may call other EJBs, resulting in a nested invocation. The primary collects in a writeset (*ws*) all the changes done in the invoked SFSBs and EBs, including creations and deletions (lines 17-21)[2]. The writeset is associated to the r_id in the ws_t table (line 20).

[2]It is important to note that the state of SFSBs is not transactional. Even if a SFSB method runs within a transaction, if the transaction aborts, the state is not undone to the previous state. Therefore, those changes are considered as committed changes. If SFSBs implement the *afterCompletion* method,

```
   // Client Side
   Data:
   c_id = X;
   r_no = 0;
1  Upon intercepting an outgoing (request)
2  |  r_id = c_id ∪ r_no;
3  |  request.attachRequestId(r_id);
4  end
5  Upon intercepting an incoming (response)
6  |  r_no++;
7  end
   // Server Side (Primary)
   Data:
   ws_t = ∅;
8  Upon intercepting an incoming (request)
9  |  EJB = request.getTargetEJB();
10 |  if EJB.requiresTransaction() then
11 |  |  startTx();
12 |  end
   |  // Now, the method is invoked
13 end
```

```
14 Upon intercepting an outgoing (response,
   request)
   |  // The method has already been executed
15 |  r_id = request.getRequestId();
16 |  EJB = request.getTargetEJB();
17 |  if EJB.hasChanged() then
18 |  |  ws = getWS(ws_t, r_id);
19 |  |  ws = ws ∪ EJB;
20 |  |  store(ws_t, r_id, ws);
21 |  end
22 |  if EJB.requiredTransaction() then
   |  |  // Complete the transaction
23 |  |  if response.isMarkedRollback() then
24 |  |  |  rollbackTx();
25 |  |  else
26 |  |  |  commitTx();
   |  |  |  // Multicast changes
27 |  |  |  if ∃r_id ∈ ws_t then
28 |  |  |  |  ws = getWS(ws_t, r_id);
29 |  |  |  |  multicast(r_id, ws, response);
30 |  |  |  end
31 |  |  end
   |  |  // Clean data structures
32 |  |  remove(ws_t, r_id);
33 |  end
   |  // The response is returned to the caller
34 end
```

Figure 3.3: One request transactions replication protocol (Client/Primary)

Before returning the response to the client, the primary completes the transaction (lines 22-33). If transaction commits successfully, the primary multicasts a message with the writeset and the response to the backups and itself using FIFO order (lines 27-30). Therefore, at most one message is sent to the backups per client invocation. Uniform multicast guarantees that if the primary has delivered that message, the backups will also deliver it. FIFO order allows the backups to commit transactions in the same order as the primary. If no EJB is modified or the transaction is marked to rollback (line 24), the primary does not send any message. In any case, the writeset table is cleaned using the r_id (line 32). Finally, the primary returns the response to the client.

The pseudocode of the backup part of the protocol is depicted in Figure 3.4. The backup replicas process the messages from the primary applying the changes in the writeset in the delivery order (line 39). Finally, the backups store the response returned to the client in the *last_resp_table* associated to the request identifier of the client (line 40). The backups update the last response stored in the table upon a new message is received from the same client.

In figure 3.5 is shown an example of the interactions considered in this protocol. When the client application invokes the transactional method *TxMethod1()* on *EJB A* (1), the application server creates a transaction (the dotted lines represent the transactional contexts). After the method invocation finishes (2), the transaction is completed and the possible changes on the state of the *EJB A* are replicated. Finally the response is returned to the client. Later on the

the EJB container executes that method after committing/aborting the transaction. This may change the state of the EJB. In this case, the primary also sends those changes in the writeset.

```
// Server Side (Backup)
Data:
last_resp_t = ∅;
Upon a multicast message is delivered (msg)
    r_id = msg.getRequestId();
    ws = msg.getWS();
    response = msg.getResponse();
    applyWS(ws);
    update(last_resp_t, r_id, response);
end
```

Figure 3.4: One request transactions replication protocol (Backup)

application code, the client invokes *TxMethod2()* of *EJB A* (3). This invocation includes a nested invocation to *Method1()* of *EJB B* (4 and 5). The execution of *Method1()* in *EJB B* is done in the context of the same transaction. When the transactional invocation initiated by the client finishes (6), the possible changes in the state of *EJBs A* and *B* are replicated and the response is returned to the client.

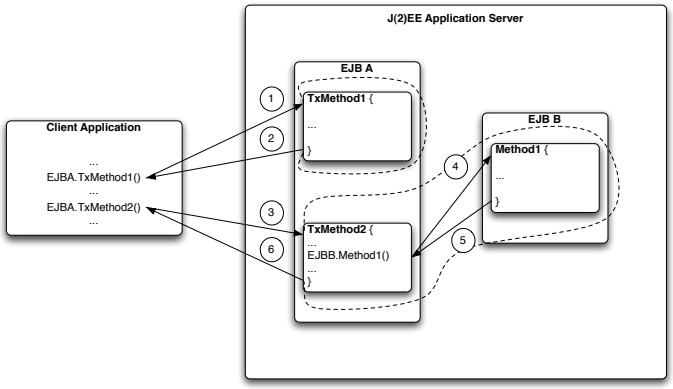


Figure 3.5: One request transactions example

3.4.1.2. Failures

A replica is considered crashed if the application server, the database or both fail. When a failure is detected, the GCS informs the rest of the replicas about the failure delivering a new view excluding the faulty replica. Then, the application server and the database, are automatically shutdown (fail-stop).

In order to failover, the client stub needs to know a list of the available replicas. The client stub receives the view of the members of the cluster from the primary. Each view has a unique identifier (*view id*). The client stub attaches the view id to each request. If the primary detects that the view id of the client is not equal to the current view id, it piggybacks the new view to the response.

At the server side, when a failure occurs in a backup replica, nothing has to be done in the

failover phase. However, if the primary fails, the first step is to chose one of the backup replicas deterministically to become the new primary (e.g. the replica with the smallest identifier in the view). Strong virtual synchrony guarantees that all the current members will have delivered all the messages sent before the failure (in the previous view). Uniformity guarantees that even if a group member delivers a message and fails immediately after, the rest of the members will deliver that message. Since strong virtual synchrony is assumed and messages are uniformly multicast, when the new view is delivered, all the backups belonging to the new view have delivered the same set of writesets than the old primary delivered. Thus, anyone of them can take over as the new primary.

Before starting to process new client requests, the new primary must achieve the same internal state than the old primary had when it failed. This means to apply the changes in the EJBs stored in the writesets received from the old primary. If those changes are not applied by the time the view change is delivered, the processing of new client requests will be delayed until all outstanding changes are applied.

If the primary fails while processing a request, the client stub receives an exception and the client resubmits the request to another replica. That replica may or may not be the new primary. If the chosen replica is the new primary, it will process the request; otherwise it will reply with a message indicating the new primary replica. Due to the use of uniform multicast, if the primary sent the message with the changes (writeset and the associated response) and delivered that message before failing, the rest of the replicas also delivered the message and also have the writeset and the response. If the primary did not deliver that message, uniformity guarantees that the rest of the replicas will either all deliver the message or none. In the former case, the new primary has delivered the changes produced by that request. When the client resubmits the request, the new primary accesses the last_resp_t table using the request identifier (r_id), detects that the request is a duplicate (the r_id already is in the results table) and sends the cached response to the client. In the latter case, none of the backups is aware of that request because the primary failed before the corresponding changes were delivered. When the new primary accesses the last_response_t table, it does not find the r_id, so it executes the request.

Note that the primary does not multicast any message if a request does not update any SFSB or EB ("read-only" request). Therefore, when a client resubmits a "read-only" request, the new primary is not aware of that request (the r_id is not found in the last_resp_t table) and must execute it.

3.4.1.3. Recovering and Adding New Replicas

In order to provide high availability to stateful applications, a cluster with N replicas ($N > 1$), allows $N - 1$ crash failures at the most. A cluster with a primary-backup scheme must always have at least one replica (the primary) processing client requests. As replicas fail in time, the number of backup replicas decreases. When the primary fails and there are no backups to failover, the application is not available anymore. Thus, the cluster must provide a mechanism to add new replicas and to recover those that have failed. The available replicas in the cluster must transfer the state to the new replicas.

The state transfer can be performed either offline or online. If the state transfer is done offline, the cluster stops the processing of new client requests, breaking the high availability. Online state transfer does not have this drawback, so the cluster can continue processing client

requests whilst new replicas are being added. Thus, a high available cluster must support online state transfer.

In a cluster with a primary-backup scheme, new or faulty replicas are re-incorporated as backup replicas. When a replica is added, its internal state must be synchronized with the state of the other replicas. Since a replica consists of an application server and a database instance, this means that not only the state of the EJB components must be transferred but also the data stored in the database.

Online state transfer can be done from the primary replica or from the backups. If the primary is the only replica in the cluster, the state transfer must be done from it. However, if there is at least one backup, for performance reasons it is better to transfer the state from one of the backup replicas (The primary is the only replica that processes client requests. The backups just apply the changes).

The primary replica can perform the online state transfer when it is not processing any request. After the new view is delivered, the new replica starts receiving the messages multicast by the primary. These messages are enqueued in the new replica until its internal state is synchronized. If the primary sent messages to the backups in between the view change delivery and the completion of the state transfer in the new replica, the primary must inform the new replica about which messages are already applied in the transferred state. The new replica must discard those messages from its queue. Finally, it can start processing the rest of enqueued messages received from the primary.

When the state transfer is done from a backup replica, the backup stops applying locally the changes of the messages delivered after the view change. As the new replica does, the backup enqueues these messages. Then, the backup starts transferring its state at that point in time to the new replica. When the state transfer is completed, the backup continues applying the changes in the enqueued messages. In this case, the new replica does not need to discard any message. Once it has received and installed the state, it can apply the changes contained in the messages it enqueued from the primary.

3.4.2. Client-Demarcated Transactions

Clients and session beans can demarcate transactions that may bracket several invocations to the application (N requests/1 transaction pattern). Thus, client-demarcated transactions contain the changes in the state of EJBs produced in all the invocations. As in the previous protocol, during an invocation, the business logic of the EJB may invoke other methods on other EJBs. This protocol also considers that these invocations may be executed by an independent transaction (N requests/M transactions pattern). However, if the main transaction aborts in this scenario, the protocol does not guarantee the global consistency of the application because the already committed transactions can not be aborted. The replication protocol takes into account the transaction demarcation actions performed explicitly by the clients (begin, commit, rollback). This is required to associate the changes performed in each invocation to each particular transaction of the client. The changes are multicast when each transactional client invocation is about to complete (eager replication).

The protocol works as follows; the primary collects the changes in EJBs produced by each client invocation in a writeset that is associated to each transaction. When the client invocation is about to finish (just before sending the response to the client), the primary multicasts the changes stored in the writeset. Writesets may contain two kinds of changes:

committed (produced by invocations to EJB methods that demarcate their own transaction) and uncommitted (produced by invocations to EJB methods that are executed in the context of the client-demarcated transaction). When the backups receive the writesets, they apply the committed changes and delay the application of uncommitted changes until they receive a message indicating the end of the transaction.

3.4.2.1. Protocol Details

The pseudocode of the protocol for the client side and the server side of the primary replica is shown in Figure 3.6.

The client side of the protocol is similar to the previous protocol. The client attaches to each EJB invocation a request identifier (*r_id*) that includes the client identifier (*c_id*) and a request number (r_no). When a client invocation triggers the beginning of a transaction, the primary creates a new transaction using the JTA and generates a transaction identifier (*t_id*). The primary attaches the t_id in the response returned to the client (line 2). From this point on, the client also includes in the request identifier the t_id (line 3).

The EJB container of the primary uses the r_id to index the access to different tables; a transaction table (*tx_t*) to associate a particular client with a transaction; a writeset table (ws_t) that associates each transaction with its writeset; and finally a table that stores the last responses sent to each client in order to deal with possible failures (*last_resp_t*).

After the primary creates a new transaction for a particular client (lines 15-18) and before returning the response to the client, the primary multicast a (*begin message*) to inform the backups about the new transaction (lines 39-43). Moreover, if the transaction is not finished in the current invocation, the transaction is suspended before returning to the client (line 40)[3]. When the primary receives a request within a particular transaction, it accesses the transaction table to resume the associated transaction and then, it processes the request (line 21). Then, the primary stores the changes in stateful EJBs in the writeset and multicasts them together with the response when the invocation is about to finish (lines 44-63). When the client commits/rolls-back a transaction, the primary resumes the associated transaction (line 32), invokes the commit/rollback operation on the transaction manager (lines 65/70), and multicast a *commit/rollback message* with the t_id (lines 66/71). Then, the outcome of the transaction is returned to the client and the data structures associated to the concrete transaction are cleaned (lines 67/72).

When an EJB method is invoked, it may start a new independent transaction[4] (lines 23-26). It is important to note that this independent transactions may cause inconsistencies in the application even if it is not replicated. For example, if an independent transaction commits but the client-demarcated transaction aborts, it is not possible to undo the changes committed by the inner transaction. The changes on EJB components produced in the context of the independent transactions are included in the writeset of the client-demarcated transaction. This implies that a writeset can carry changes from several transactions and that those changes can be both uncommitted and committed (lines 48-52). Uncommitted changes are the changes of stateful EJBs produced in the context of the client-demarcated transaction (and that are

[3] This is done by J(2)EE applications servers, even if they are not replicated, to disassociate the current transaction from the thread processing the invocation.

[4] In J(2)EE it is possible to run several transactions on behalf of a single client invocation, by suspending/resuming transactions. The enclosing transaction is suspended, and a new inner transaction is started.

```
// Client Side
Data:
c_id = X;
r_no = 0;
t_id = null;
Upon intercepting an outgoing (request)
    if t_id == null then r_id = c_id ∪ r_no;
    else r_id = c_id ∪ r_no ∪ t_id;
    request.attachRequestId(r_id);
end
Upon intercepting an incoming (response)
    if t_id == null then
        t_id = response.getTxId();
    end
    r_no++;
end
// Server Side (Primary)
Data:
ws_t = ∅;
tx_t = ∅;
Upon intercepting an incoming (request)
    r_id = request.getRequestId();
    switch request.getType() do
        case "begin"
            t_id = beginTx();
            store(tx_t, r_id, t_id);
        end
        case "invocation"
            if ∃r_id ∈ tx_t then
                resumeTx(r_id.getTxId());
                EJB = request.getTargetEJB();
                // If the method requires a new
                   Tx
                if EJB.requiresNewTransaction() then
                    suspendTx();
                    startTx();
                end
                // Now, the method is invoked
            end
        end
        // commit or rollback
        otherwise
            if ∃r_id ∈ tx_t then
                resumeTx(r_id.getTxId());
            end
        end
    end
end
Upon intercepting an outgoing (response,
request)
    r_id = request.getRequestId();
    switch request.getType() do
        case "begin"
            suspendTx();
            multicast("Begin", r_id, response);
            response.attachTxId(r_id.getTxId());
        end
        case "invocation"
            // The method has already been
               executed
            EJB = request.getTargetEJB();
            ws = getWS(ws_t, r_id);
            if EJB.hasChanged() then
                // If the method was executed by
                   an independent Tx
                if EJB.requiredNewTransaction() then
                    ws.committed = ws.committed ∪
                    EJB;
                else
                    ws.uncommitted = ws.uncommitted
                    ∪ EJB;
                end
            else
                ws.read = ws.read ∪ EJB;
            end
            // Commit the possible independent
               Tx and resume the previous one
            if EJB.requiredNewTransaction() then
                commitTx();
                resumeTx();
            end
            store(ws_t, r_id, ws);
            suspendTx();
            multicast("Invocation", r_id, ws, response);
        end
        case "commit"
            commitTx();
            multicast("Commit", r_id, ws, response);
            cleanDataStructures(tx_t, ws_t, r_id);
        end
        case "rollback"
            rollbackTx();
            multicast("Rollback", r_id, ws, response);
            cleanDataStructures(tx_t, ws_t, r_id);
        end
    end
    // The response is returned to the caller
end
```

Figure 3.6: Client-demarcated transactions protocol (Client/Primary)

still uncommitted). Committed changes comprise the changes in stateful EJBs occurred in the context of independent transactions that have been already committed.

As it occurs in the previous protocol, when each client request corresponds exactly to a single transaction, backups only need to apply committed changes. These kind of requests do not need to know which reads were performed in the database. However, when transactions

span multiple client requests, this is not the case anymore. Reads performed by uncommitted transactions at the primary are important to guarantee consistency if the primary fails, since they affect the serialization of transactions (e.g. by setting read locks). For instance, let us consider two concurrent transactions $T1$ and $T2$. $T1$ has read object X and $T2$ has just begun. Then, the primary fails. $T1$ and $T2$ are recreated on the new primary. Now, $T2$ modifies object X and commits. $T1$ performs other operations and reads X again. The value of X is different to the value read previously, though, both reads are executed within the same transaction (non-repeatable reads). If the primary would not have failed, $T2$ would block when trying to modify X until $T1$ finishes. In order to implement the same semantics for failure and failure-free scenarios, the primary in addition to the changes also sends information about database reads. Since results of reads can be very bulky, the primary sends the SQL read statement submitted by the container to the database (line 54).

The pseudocode of the protocol for the server side of the backup replicas is shown in Figure 3.7. When a backup receives a begin transaction message, it stores the information received in the transaction table (lines 81-83). Upon a backup receives writesets from the primary for a particular transaction (lines 84-87), it simply applies all committed changes and stores uncommitted changes. A backup will delay the application of uncommitted changes till it processes the commit message. When a backup processes the commit message, it applies the uncommitted changes for that transaction (line 89) and discards the information associated to the transaction (lines 90-91). If a backup receives an abort message (lines 93-96), it discards the information stored in the transaction and writeset tables (uncommitted changes and SQL read statements). After processing each message, the backup stores the result of the request in the last_resp_t table.

Figure 3.8 shows an example of the interactions considered in this protocol. First of all, the client application demarcates a transaction (1). When the invocation is captured by the application server, it creates a new transaction using the transaction manager (the dotted lines represent the transactional contexts), and multicast a *begin* message to the other replicas. When the client application invokes method *TxMethod1()* on *EJB A* (2), the invocation occurs on the previously created transaction. After the method invocation finishes (3), the possible changes on the state of the *EJB A* are replicated as uncommitted changes and finally, the response is returned to the client. Later on the application code, the client invokes *Method2()* of *EJB A* (4). This invocation includes a nested invocation to *TxMethod1()* of *EJB B* (5 and 6). This method is executed in an independent transaction, so the previous transaction is suspended. When the invocation finishes, the possible changes produced by *TxMethod1()* in *EJB B* are captured as committed changes and the previous transaction is resumed. When the transactional invocation initiated by the client finishes (7), the possible committed and uncommitted changes in the state of *EJBs A* and *B* are replicated and the response is returned to the client. Finally, the transaction demarcated by the client is completed (8). When this occurs, the transaction manager commits (such as in this case) or rollbacks the transaction, and multicast the corresponding message to the other replicas to apply or not the uncommitted changes. However, if the client demarcated transaction aborts, with this protocol the committed changes can not be undone.

```
// Server Side (Backup)
Data:
ws_t = ∅;
last_resp_t = ∅;
tx_t = ∅;
Upon a multicast message is delivered (msg)
    r_id = msg.getRequestId();
    response = msg.getResponse();
    ws = getWS(ws_t, r_id) ∪ msg.getWS();
    switch msg.getType() do
        case "Begin"
            store(tx_t, r_id, r_id.getTxId());
        end
        case "Invocation"
            applyWS(ws.committed);
            store(ws_t, r_id, ws.read, ws.uncommitted);
        end
        case "Commit"
            applyWS(ws.uncommitted);
            remove(ws_t, r_id);
            remove(tx_t, r_id);
        end
        case "Rollback"
            remove(ws_t, r_id);
            remove(tx_t, r_id);
        end
    end
    update(last_resp_t, r_id, response);
end
```

Figure 3.7: Client-demarcated transactions protocol (Backup)

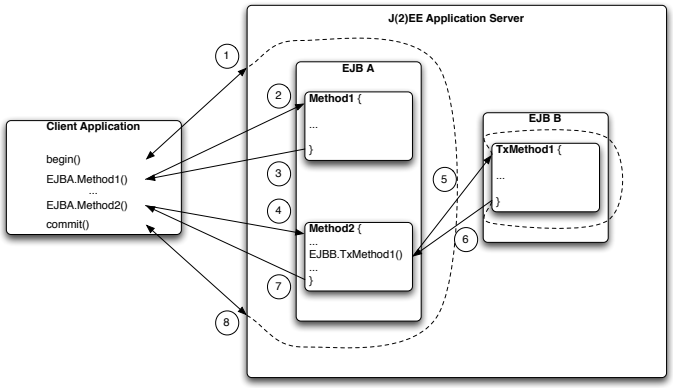


Figure 3.8: Client demarcated transactions example

3.4.2.2. Failures

The simplest failure scenario arises when the primary fails between two client invocations. In this case, the client uses the information of the current view to contact other replica. The replica will sent back an updated view with the new primary replica. Finally, the client will re-send to the new primary its last request, resuming their interaction.

When the client invokes the begin operation and the primary fails, two cases may arise. If the transaction is stored in the transaction table of the new primary, the t_id is sent back to the client. Otherwise, the new primary did not delivered the begin message in the previous view and processes the message as if were new. The same process is applied when the client invokes the commit or abort operations.

If the primary fails while running an invocation, the client stub finds out the new primary and re-submits the invocation. As it occurs in the previous protocol, if the new primary received the multicast message from the previous primary, it returns the cached response in the last_resp_t table. Otherwise, the new primary executes the request as if were new.

Moreover, when the primary fails before a client transaction completes, replicating the uncommitted state on each client invocation avoids the abortion of the client transaction. The new primary resumes the transaction execution transparently providing highly available transaction support. The process is as follows; the new primary does not process any client request until it applies the changes in all the writesets multicast by the old primary before the view change. Then, the primary checks if there are uncompleted transactions in the transaction table. For each entry in the table, the new primary creates locally a new transaction, applies uncommitted changes and executes the associated reads in the order they were sent by the old primary. This guarantees that each re-created transaction holds the same state it held at the old primary.

3.4.3. Open Nested Transactions

Open Nested Transactions (ONTs) are transactional activities –also called units of work– demarcated by client applications or by session beans. In contrast to the previous protocol, ONT activities may be nested. This can yield to arbitrarily complex ONT activity trees[5]. A top-level ONT activity may encompass several transactional sub-activities (1 request/N transactions pattern) and multiple transactional invocations to sub-activities can be done in the context of a top-level ONT activity (N requests/M transactions pattern). The changes made on these invocations are part of the sub-activity that encompasses them. Whenever a child ONT activity commits, a compensator to undo semantically the committed work is registered with its parent. This procedure is performed recursively until the top-level ONT activity is reached. This allows to compensate the previous work committed in the context of the top-level ONT activity, and thus, atomicity is guaranteed. When the top-level activity commits, the registered compensators are discarded. When an ONT activity aborts, the registered compensators are executed to undo the changes of the previously committed sub-activities.

The protocol for replicating ONTs is an extension of the previous protocol. Thus, the protocol is similar to the previous one. However, in this case, besides the changes produced in each invocation and the response returned to the client, the primary must send at the end of each client invocation, the ONT compensators of committed sub-activities.

3.4.3.1. Protocol Details

When a client demarcates the beginning of an activity, the primary intercepts the *begin* activity message and creates a top-level ONT activity using the ONT HLS. The activity identifier (*a_id*) of the activity is used to index an ONT activity table (*act_t*) that contains the

[5]EJB methods that demarcate non-ONT transactions are not handled by the protocol.

structure of an ONT activity. The a_id is returned to the client at the end of the invocation. Before returning the a_id to the client, the primary suspends the ONT activity and multicasts the *begin* message. From this point on, when a client submits a request, it attaches the a_id to the request identifier ($r\_id = c\_id + r\_no + a\_id$).

When the primary receives a client invocation, it resumes the associated top-level ONT activity and executes the request. The invocation may require a nested ONT sub-activity when a EJB is invoked. The primary registers the nested ONT activity in the ONT activity table as a child activity. If the nested ONT activity succeeds, a compensator is added in a special stack (*comp_stack*) associated to each ONT activity. The compensators will be extracted and executed if an ancestor ONT activity fails. At the end of the client invocation, the primary multicasts the request identifier, the response, the writeset (with the changes of the committed nested ONT activity/ies as committed changes and the changes in the context of the top-level ONT activity as uncommitted ones), read statements and the new registered compensators.

If the client submits a *commit* operation, the primary resumes and commits the associated top-level ONT activity, multicasts the ONT activity outcome, and returns to the client. On the other hand, if the parent ONT activity fails, the primary executes the compensators and afterwards, it collects all the EJB changes. Then, it multicasts a *rollback* message with the changes and the ONT activity outcome. After delivering this message, it returns the outcome to the client. In both cases, commit and abort, the a_id is removed from the ONT activity table.

Upon receiving the *begin* message, the backups store the information received in the ONT activity table. When a backup receives a message with changes, it proceeds as in the previous protocol: it applies the committed changes and stores uncommitted changes, reads, compensators and corresponding response for each *r_id* . If the backup receives a *commit* message, it applies all uncommitted changes. On the other hand, if the backup receives a *rollback* message, it applies the changes corresponding to the execution of the registered compensators (the committed changes attached to the message received) and discards the uncommitted changes and reads previously stored.

Finally, figure 3.9 shows an example of the interactions considered in the ONT protocol. Initially, the client application demarcates the top-level ONT activity (1). When the invocation is captured by the application server, it creates a new ONT activity using the ONT high level service (the dotted lines represent the activity contexts), and multicast a *begin* message to the other replicas. Then, the client invokes *Method1()* of *EJB A* (2) in the context of the top-level ONT activity. The changes performed by this method on the state of EJB A are considered as uncommitted changes. However, this invocation includes a nested invocation to *Method1()* of *EJB B* (3 and 4). This method is executed in a child ONT activity, nested in the top-level activity. When the invocation finishes, the possible changes produced by *Method1()* in *EJB B* are captured as committed changes and a compensator is registered in the parent activity. When the invocation initiated by the client finishes (5), the possible committed and uncommitted changes in the state of *EJBs A* and *B* are multicasted joint with the compensators and finally, the response is returned to the client. Later on, the client application invokes *Method2()* on *EJB A* (6), the invocation occurs also on the context of the top-level ONT activity. After the method invocation finishes (7), the possible changes on the state of the *EJB A* are replicated as uncommitted changes and finally, the response is returned to the client. Finally, the top-level ONT activity demarcated by the client is completed (8). When this occurs, the application server commits (such as in this case) or rollbacks the ONT

activity using the ONT high level service, and multicast the corresponding message to the other replicas to apply or not the uncommitted changes. In this case, if the client demarcated top level ONT activity aborts, this protocol allows to undo the previously committed changes by means of the compensators.

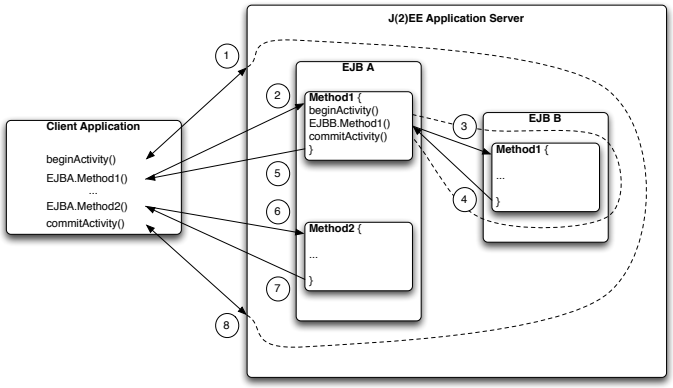


Figure 3.9: Open nested transactions example

3.4.3.2. Failures

The failover process is quite similar to the one of the previous protocol. It can be summarized as follows; When the new primary finishes processing messages from the previous primary, it re-creates all unfinished top-level ONT activities contained in the ONT activity table. Then, it associates the registered compensators, to each activity. Finally, it applies uncommitted changes, and executes the reads performed on behalf of the ONT. When this process completes, the new primary can resume request processing.

3.5. Evaluation

This section describes the evaluation of the replication protocols. The experiments have been executed in a high available cluster in which each replica consists of 2 GHz AMD's dual-processor machines with 512 MB of RAM running Red Hat Linux 9.0 operating system. The software used for the different tests includes the JBoss v.3.2.3 application server (with the ADAPT replication framework when required) [Reda], PostgreSQL v.7.3.2 DBMS [Posb], JGroups [JGr] GCS and JASS [Obje], an implementation the activity service specification developed to test the open-nested transactions (ONTs) replication protocol. In all the experiments, the clients generating the load injected into the applications, the application server instances and the DBMSs were run in separate nodes.

The protocols that have been evaluated include the one for client-demarcated transactions and the open-nested transactions. The one-request transactions protocol has not been evaluated because is included implicitly in the other protocols. The evaluation of the overhead introduced by the client-demarcated transactions protocol has been analyzed using a J(2)EE transactional application included in the ECPerf benchmark [Sun03a]. For the evaluation of

the ONTs protocol, an application that creates nested activities has been developed ad-hoc. In this latter case, a prototype of the Activity Service for J(2)EE [Sun06b] has been used to implement a high level service (HLS) that models the behavior of ONTs. In both applications, client requests need to be simulated in order to stress the system. To accomplish this task, a client application to inject load into the application has been used.

A non-replicated standard configuration of JBoss and a cluster configuration –based on the standard facilities that JBoss provides for clusters– have been used as the baselines for the experiments. However, none of them provides the availability and fault-tolerance guarantees that the protocols described here do. In the standard cluster configuration provided by JBoss, it is assumed that there is a single shared database among all the application server instances. The replication protocol of JBoss is based on a primary-backup scheme. However, it is not transaction aware and multicasts the state of an stateful session bean every time is modified.

3.5.1. Client-Demarcated Transactions Replication Protocol

The evaluation of the replication protocol for client-demarcated transactions has been done using the ECPerf [Sun03a] benchmark. This benchmark allows to measure the performance and scalability of J(2)EE application servers. It simulates a supply-chain management (SCM) process scenario by means of an J(2)EE application that fits to that business environment. The application defines four application domains representing the major areas of a fictitious organization: corporative, order entry, supply chain management and manufacturing. ECperf measures the throughput in *BBops* (*benchmark business operations*) per minute. BBops are the sum of the number of transactions of the order entry application and the number of work orders the manufacturing application generates. All the results of the experiments are obtained over the steady phase of the test, ignoring the outputs of the ramp up and cool down phases.

ECPerf includes a multithreaded client application that injects load to the SCM application. Each thread simulates a client of the application and injects requests synchronously, that is, a new request is not injected until the response of the previous one has been received. The main parameter varied in the tests has been the *injection rate* (IR), which models the amount of load injected to the system. The number of clients is five times the IR in the order entry application, and three times the IR in the manufacturing application. The ECPerf target determines for a given load, the conditions that the performance metrics of the system should fulfill. It specifies a maximum response time for all the requests (2 seconds for the order entry domain and 5 seconds for the manufacturer domain) and also that the response time corresponding to the percentile 90% is no more than a 10% of the average. This guarantees that the distribution of the response times does not have a long tail, and that most of the requests have a similar response time.

We have stressed the ECperf application to quantify the overhead introduced by our replication protocol. The results obtained when performing the tests on our protocol (*HA Replication*) have been compared with these two baseline configurations: a non-replicated JBoss (*Non-Replicated JBoss*) and the standard primary-backup configuration of a cluster of JBoss application servers (*JBoss Primary-Backup*). For each one of these three configurations, we have obtained results of the average response time for the client requests and the ECPerf's throughput. We also have measured the number and size of the messages sent by the protocols in each configuration. Table 3.1 summarizes the different configurations used in the experiments (All configurations include one node for running the clients that generate the load

into the application.).

Configuration	# Sites	# App. Server instances	# DB instances
Non-Replicated JBoss	3	1	1
JBoss Primary-Backup	4	2	1
HA Replication	5	2	2

Table 3.1: Tested Configurations

Figure 3.10 shows the overall average response time for the order entry transactions (the ones with stricter response time) under ECPerf. As expected, non-replicated JBoss offers the lowest response time, since it does not incur any overhead due to replication. Until an IR = 10, the response time of our protocol (HA Replication) is similar to the one of the standard JBoss clustering (JBoss Primary-Backup) and the Non-Replicated JBoss. From an IR of 10 to 20, the overhead of replication has a noticeable impact on the response time of our protocol becoming higher than the one of JBoss primary-backup. However, the response time is still within the limits admitted by ECPerf for order entry transactions (2 seconds). This means that for moderate loads the overhead of our replication protocol is negligible and only for high loads it results in an increased, although still reasonable, response time.

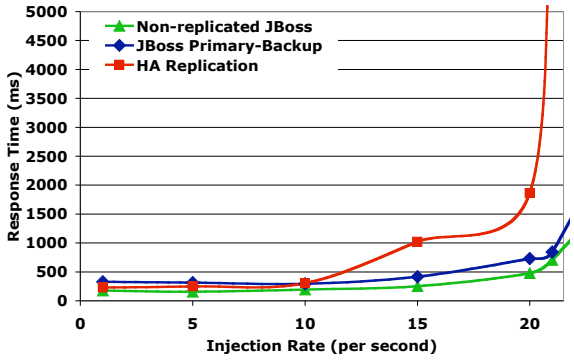


Figure 3.10: ECperf results: Average response time

Figure 3.11 shows the maximum throughput with respect to the ECPerf target. All configurations fulfilled the ECPerf target until an IR = 20. HA Replication saturates for higher IRs. JBoss Primary-Backup and the non-replicated JBoss accomplished the target for an IR = 21. This means that with respect the ECPerf target, our replication protocol has a loss of throughput of about 5%. It is also worth to notice that the non-replicated JBoss degrades more gracefully under saturation. In particular for an IR of 21, our replication protocol was far from reaching the ECPerf target, whilst JBoss Primary-Backup was not that far for an IR of 22. This means that the results in terms of the ECPerf target are very promising for the replication protocol proposed. The additional cost of HA Replication is also natural since it is transaction aware and takes into account both, the state of persistent components (Entity Beans, EBs) and components holding session state related to each client (Stateful Session Beans, SFSBs), whilst the standard JBoss Primary-Backup clustering only replicates session components (It does not replicate EBs and is not transaction aware).

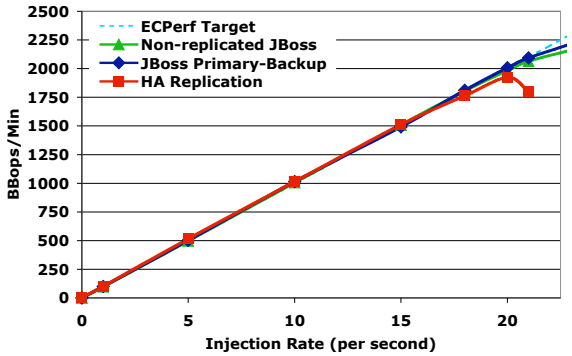


Figure 3.11: ECperf results: Throughput

Table 3.2 shows the number of messages sent from the primary to the backups and their average size for the order entry domain with IR = 10. The results show that JBoss Primary-Backup and HA Replication send a similar number of messages for the order entry part of the application, though, JBoss Primary-Backup does not replicate EBs. JBoss Primary-Backup sends a message every time a SFSB is modified, and also sends internal control messages. On the contrary, HA Replication sends replication messages only when each method call is about to be finished and changes on SFSBs and/or EBs have been produced. This fact also explains the higher average sizes of messages measured for the HA Replication.

Order entry domain (IR = 10)		
System	**Number of Msgs.**	**Avg. size(bytes)**
JBoss Primary-Backup	5169	797
HA Replication	5261	2073

Table 3.2: Message statistics in order entry domain

Table 3.3 also shows the results for an IR = 10 and both, the order entry and the manufacturing domains. The number of messages sent by the HA Replication protocol increases when the manufacturing domain is taken into account. It sends about three times more messages than JBoss Primary-Backup approach. In both application domains, the messages are also three times larger than the ones of JBoss. This extra overhead is reasonable taking into account that JBoss only replicates the state of SFSBs that are scarcely used in ECPerf (only the shopping cart of the ECPerf's application is modeled as an SFSB) and does not replicate persistent state (EBs). This means that when a stateless session bean (SLSB) is invoked and only accesses EBs, JBoss replication does not send any message, whilst our protocol sends a message with the EBs updated.

3.5.2. Open Nested Transactions Replication Protocol

The goal of this experiment is to evaluate the overhead of our replication protocol for applications with long running activities. In order to do this, the J2EE Activity Service specifi-

Order entry and manufacturing domains (IR = 10)		
System	**Number of Msgs.**	**Avg. size(bytes)**
JBoss Primary-Backup	5406	786
HA Replication	17183	2095

Table 3.3: Message statistics in order entry and manufacturing domains

cation and the open nested transaction model (ONT) have been implemented and integrated into the JBoss application server[6].

At the time of performing the evaluation, we are not aware of any benchmark to evaluate advanced transaction models. So, a custom benchmark has been built to perform the experiments. The benchmark consists of a shopping cart application derived from the one in the ECperf order entry application domain. All the ECPerf infrastructure to inject the load and evaluate whether the target was fulfilled has been reused in the new benchmark. The load generator for the shopping cart application is an adapted version of the one provided by ECPerf. This allows to generate the client requests as ECperf does. The injection rate (*IR*) is the parameter used for determining the experiment load. That is, this benchmark has same level of strictness as ECPerf.

An schema of the benchmark's shopping cart application is depicted in Figure 3.12. The application works as follows; Every client adds between five and fifteen items to the shopping cart. The first time a client adds an item, an ONT representing the cart is created (*Top-level TX*). Every time that a client adds an item to the cart, a child transaction of the top-level ONT is started (*Middle TX*). Within this transaction, the application updates the client credit and stock of the item accordingly. Each of these two actions is a nested ONT transaction (*Leaf TX*). When any of these ONTs commits, a compensating action is registered in the parent transaction (*Middle TX*). The compensating action for the item increases the quantity of the item in the stock. The one for the customer increases the customer credit. If the customer has enough credit and the selected item is available in the stock, the item is added to the cart, and the middle ONT registers the compensators with the top-level ONT. On the other hand, if one of the checks fails, the middle transaction aborts and the registered compensators are executed. Finally, the client decides whether to buy the contents of the cart (the top-level ONT commits or aborts). If the top-level ONT commits, the compensators are forgotten. Otherwise, the compensators are executed by the top-level ONT undoing the changes previously made by its children transactions.

In this evaluation, two different configurations have been used: one using the standard JBoss extended with the J2EE Activity Service (*Non-Replicated JBoss*) and the other using the replication protocol (*HA Replication*). The former uses three nodes, one for JBoss, one for the database and another one for clients. HA Replication uses five nodes: one for the clients, two nodes for each of the two JBoss, and two nodes for each of the two databases. The JBoss clustering configuration was not used this time, since JBoss clustering does not support the replication of the J2EE Activity Service. Table 3.4 summarizes these two configurations.

The shopping cart application has been stressed to quantify the overhead of the open-nested transaction replication protocol with regard to a non-replicated version of the application. We

[6]The implementations of the J2EE Activity Service and the ONT model are available at ObjectWeb as part of the JASS open source project: http://forge.objectweb.org/projects/jass/

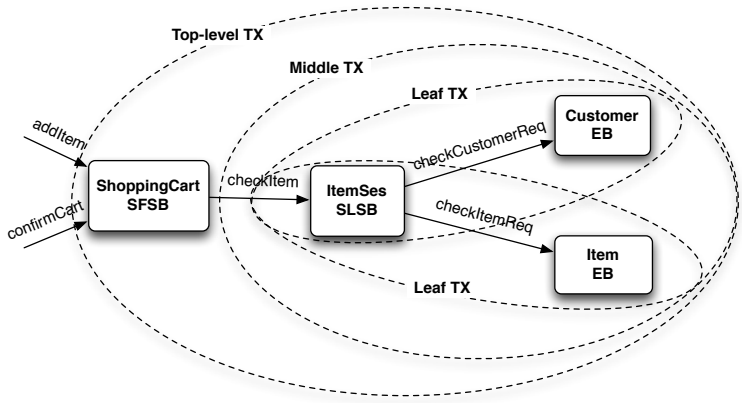


Figure 3.12: Schema of the application used as benchmark for the ONT protocol

Configuration	# Sites	# App. Server instances	# DB instances
Non-Replicated JBoss	3	1	1
HA Replication	5	2	2

Table 3.4: J(2)EEAS Benchmark Configurations

have obtained results of the throughput and we also have measured the CPU usage in each configuration for different generated loads.

The throughput of both the, Non-Replicated JBoss and the HA Replication protocol are very close (Figure 3.13). Both are able to reach the target till an IR of 7. The throughput beyond an IR of 7 keeps similar until an IR of 10. At that point, our replication protocol degrades very fast, whilst the non-replicated JBoss exhibits a more or less flat curve before falling. The reason for the faster degradation of our replication protocol is the overhead produced by the EB replication and the structure of ONTs that, although is moderate, once the server is saturated it becomes noticeable.

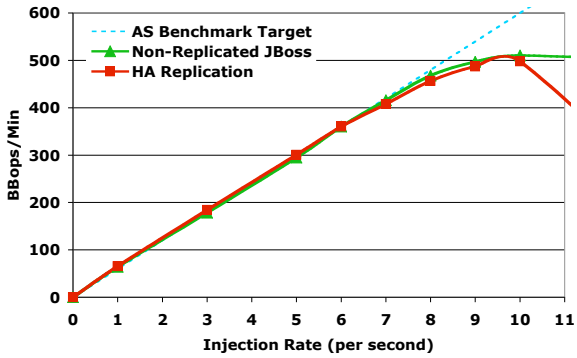


Figure 3.13: Activity service evaluation results: Throughput

Table 3.5 shows the CPU usage for two different IRs (3 and 8). When low loads are

injected (IR = 3), the throughput of both configurations is the same (See Figure 3.13), but the CPU usage of the HA Replication is slightly greater than the Non-Replicated JBoss due to the overhead of the replication mechanism. For high loads, near the saturation point (IR = 8), the CPU in the HA Replication protocol is almost saturated with an usage of 99% whilst the Non-Replicated JBoss still exhibits an acceptable CPU usage.

Configuration	CPU Utilization (IR = 3)	CPU Utilization (IR = 8)
Non-Replicated JBoss	27%	60%
HA Replication	42%	99%

Table 3.5: CPU usage in J(2)EEAS benchmark

In summary, our replication protocol shows a good behavior for long running client interactions. The results obtained are very promising for other applications using long running transactions because the overhead of replicating ONTs is not perceptible in terms of throughput.

3.5.3. Failover Evaluation

This experiment measures the time that is necessary to complete the failover process. The failover process has been tested with the client-demarcated transaction protocol. It starts at the very instant at which a replica in the cluster detects the failure of the previous primary. Since the protocol is based on group communication, the failure detection of the primary is instrumented by means of view changes. When the primary fails, the underlying group communication infrastructure delivers a message containing the view with the new membership. The backup that appears first in the view takes the role of new primary and the failover process is triggered.

The length of the failover process mainly depends on two aspects:

- The time needed to recreate each uncommitted transaction (and the activity tree if the protocol is the open-nested transactions).
- The number of updated entity beans (EBs) that have not been committed yet.

Upon failover, the new primary re-creates each uncommitted transaction (and the activity tree in case of the open-nested transactions protocol) and applies the updates on EBs stored as uncommitted changes. When these steps are finished, the failover process is finished and the new primary can process new client requests.

The number of uncommitted transactions depends on the number of concurrent clients. In order to quantify the cost of the failover, the experiment measures the failover time for an increasing number of concurrent clients and uncommitted EBs per client (1, 5, and 10 EBs per client). The clients are emulated through a load generator created specifically for the failover evaluation.

Figure 3.14 shows the failover time obtained for an increasing load. There are three curves corresponding to a varying number of uncommitted EBs per client. The figure shows that even for large loads (100 clients) and the highest number of uncommitted EBs per client (10 EBs), the time taken by the failover is quite affordable (about 250 ms). In this worst case scenario the new primary has to recreate one hundred transactions (one per client) and apply the changes

of 1,000 uncommitted EBs during failover. The failover time is almost independent of the number of EBs for small loads (until ten clients). When the number of clients increases, the number of EBs becomes the dominant factor of the failover time. It can be concluded that the failover time is quite reasonable even for high loads. As far as the failover time is smaller than the time failure detection takes, the failover overhead is acceptable. Failure detection time is typically around 1 second to prevent false failure suspicions when the system is overloaded, so failover shows good performance.

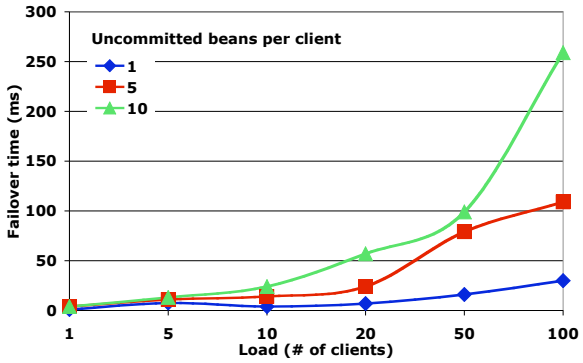


Figure 3.14: Failover time

3.6. Conclusions

This chapter has presented a novel approach to attain high availability using primary-backup replication in a cluster based on the J(2)EE platform. This approach offers several innovations to the replication multi-tier architectures. The first one is that the approach makes the J(2)EE replication, transaction and activity aware. Another novelty is that it provides highly available support for transactional applications and long running activities that can span multiple client interaction patterns with the application server. This is important in many applications such as workflow systems, OLAP. . . , in which part of their state and part of the processing is delegated to a J(2)EE application server. For these applications, re-execute a transactional operation is either very complex or not possible at all. With the proposed solution, running transactions and long running activities continue their processing despite replica failures transparently to client applications. Finally, the protocols also provide the consistent replication of the application server and the database as a single unit, what is called vertical replication. Previous approaches either replicate the application server (and forced to use a shared database) or replicate the database tier. The major drawback of these approaches is that the non-replicated tier may become a single point of failure.

The protocols presented in this chapter have been implemented and evaluated using a commercial J(2)EE application server (JBoss). The protocol for client demarcated transactions has been evaluated using the transactional application included in the ECPerf benchmark for J(2)EE application servers. The results have shown that the overhead of the protocol is very reasonable, even when compared to approaches that do not replicate the database tier and

do not provide highly available transactions. Moreover, the performance evaluation of the protocol for long running activities shows that the overhead introduced is hardly negligible.

CHAPTER 4

High Availability and Scalability in Multi-Tier Architectures

What does not kill us, makes us stronger.

– Friedrich Nietzsche

The new vision of clusters and enterprise grids not only requires high available infrastructures, but also infrastructures that provide high performance to applications. A cluster that can be scaled allows to improve the performance of applications by adding new replicas to it. Moreover, in clusters, replicas are susceptible to fail and need to be replaced. These scenarios imply that, when a new replica is added or re-incorporated to a cluster, a recovery process must be triggered.

This chapter introduces a replication and a recovery protocol that jointly provide consistency, high availability and scalability to multi-tier applications running in high performance clusters.

4.1. Introduction

In current enterprise data centers there are many applications based on the multi-tier architecture approach. Nowadays, these applications are demanding higher levels of availability and scalability to guarantee the increasing quality of service (QoS) that is required by current businesses. Moreover, the new extreme processing transaction (XTP) applications require much more resources than traditional transaction applications. They are designed, developed,

deployed, managed, and maintained on clusters and/or distributed grid computing networks because they must support higher loads from clients. Thus, the underlying cluster infrastructures should provide high availability and scalability to these applications in a transparent fashion.

The previous chapter has shown how high availability and fault-tolerance can be provided to multi-tier applications using replication. However, attaining a consistent solution to provide not only high availability but also scalability is still an open problem. Many of the recent replication approaches do not address the scalability problem and just focus on availability (e.g. FT-CORBA [NMM02a, ZMM05], J(2)EE [WKM04, WK05] and the protocols presented in the previous chapter for the J(2)EE framework).

In this chapter it is proposed a new replication protocol based on the update-everywhere approach that provides a consistent, high available and scalable solution for multi-tier applications running in high performance clusters. The replication protocol is embedded in a multi-version cache at the middle tier, and uses the vertical replication scheme to replicate both, the middle tier and the data tier, at the same time. Thus, the solution is fully implemented within the application server on top of an off-the-shelf database and avoids that the database becomes a single point of failure and a bottleneck. The cache provides snapshot isolation as the correctness criterion for the execution of concurrent transactions. Using a single replica, the cache improves the performance of applications and provides caching transparency (its semantics is the same as a system that does not use caching). Moreover, in a high performance cluster, the cache provides *one-copy snapshot isolation*, that is, the replicated system behaves as a consistent non-replicated system providing snapshot isolation guarantees.

The success of snapshot isolation is such that it has substituted serializability as the highest isolation level in current DBMS implementations: Oracle, PostgreSQL, MS SQL Server... Serializability has contention problems due to read-write conflicts that reduce the potential concurrency and the performance of the system. Thus, current application servers providing serializability to guarantee the isolation of transactions are incorrect when used with these databases. To the best of our knowledge this protocol is the first one to provide a scalable and integrated solution for the replication of both the application server and database tiers. It is also the first protocol that implements snapshot isolation for the application server cache, so it works properly with a database based on snapshot isolation.

When the replicas fail in a cluster, the high availability and the performance of the whole cluster may be compromised. Moreover, these possible failures should not interfere with the work done in the cluster. The most extreme case occurs when all the replicas become unavailable because there is no mechanism to re-incorporate the failed replicas to the cluster. However, apart from this case, there are other important scenarios to deal with. For example, management operations may also affect the availability and performance of the clusters. The system administrators stop some replicas from time to time in order to perform management or tuning operations in both, hardware and software infrastructures. When the maintenance is finished, the replicas should be re-incorporated to the cluster. In high performance clusters with high availability, new issues arise. On the one hand, when a replica fails, the applications are still accessible for the clients. However, as the load is distributed among the replicas following an update-everywhere scheme, the performance of the system may be affected in a negative way if the load is high and the current replicas are overloaded. On the other hand, some applications may require an improvement of their response times, what requires to scale out the cluster. In both cases, the systems administrators must add new replicas to the cluster

for a better distribution of the work to be done.

Thus, in addition to the replication protocol, this chapter presents a novel online recovery protocol that complements the replication protocol providing a mechanism to rejoin failed or new replicas to the cluster, what allows to manage the scale out of the cluster. The recovery protocol operates at the application server level and exploits the data hold by the replication protocol in order to deliver the required information to the failed or fresh replicas during the recovery process. This approach enables to recover the whole state of the replica in parallel with the help of the working replicas. In this way, any working replica with an up-to-date state can contribute to the recovery process without suffering excessive performance degradation and therefore resulting in a high performance online recovery approach. At the end of the recovery process, the protocol guarantees that the state of the new replica rejoined to the cluster is consistent at the middle and the data tiers.

The replicated multi-version cache, the replication protocol and the recovery protocol have been implemented and integrated into JOnAS [Objf], an open source J(2)EE application server. The performance of the implementation has been evaluated with the industrial benchmark for J(2)EE application servers, SPECjAppServer[1] [Sta04].

The chapter is structured as follows; Section 4.2 resumes the required background to understand the protocols presented here. Section 4.3 presents the system model used. The replication protocol is presented in Section 4.4. Then, the online recovery protocol is described in Section 4.5. The evaluation is shown in Section 4.6. Finally, Section 4.7 presents the conclusions.

4.2. Background

The replication protocol presented in the next section provides consistent high availability and scalability to transactional J(2)EE multi-tier applications deployed in clusters.

JOnAS [Objf], an open-source J(2)EE application server, has been modified to implement and deploy the replication protocol. The protocol fully embeds the replication logic in a new JOnAS service (*Replication Module*). Each application server communicates with its own DBMS through *JDBC*. Figure 4.1 shows an schema of the components and services of J(2)EE that are present in a replica. The figure shows how these elements are tied together.

In J(2)EE, transactions are coordinated by the *transaction manager*. This service is accessed using the *Java Transaction API* (JTA). Transactions can be demarcated either explicitly (*bean managed transactions*) or implicitly (*container managed transactions*, CMTs). With CMTs, the application server intercepts the component invocations and demarcates transactions automatically. The protocol uses CMTs and synchronizes the transactions at the application server level with the transactions at the database level.

The *EJB container* is in charge of managing the EJB components. When clients perform transactional invocations, the application server stores the changes in EJB components produced in each transaction in a writeset. So, each writeset is associated to a particular transaction. Only stateful components need to be replicated. Thus, only stateful session beans (SFSBs) and entity beans (EBs) are considered by the protocol. Moreover, EBs are

[1]SPECjAppServer is the new version of the ECPerf benchmark for J(2)EE application servers that was introduced in the previous chapter. At the time of writing this Ph.D. Thesis, SPECjAppServer 2004 is the latest version of the benchmark.

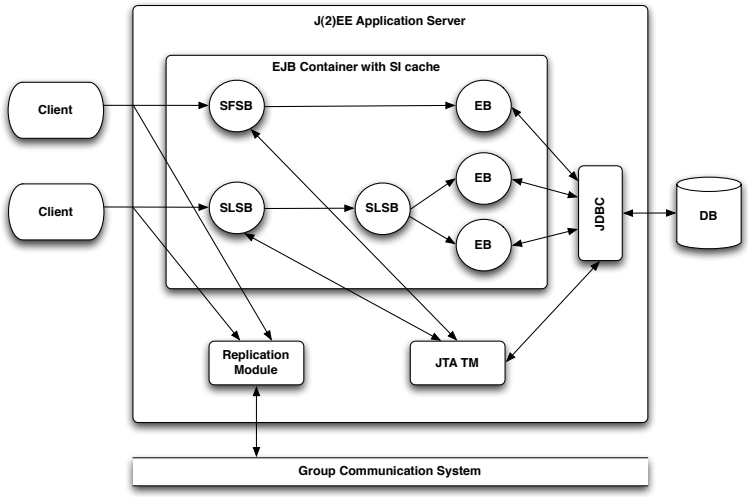


Figure 4.1: Main J(2)EE components and services and their relationships

used to cache an object oriented view of the database items used by the applications. In order to keep this view consistent with the database, application servers implement a concurrency control policy to manage the access to the components in the cache. They typically provide serializability as correctness criteria, implemented via locking or optimistic schemes, since databases have relied on serializability for a long time. However, today many databases provide snapshot isolation as the highest isolation level (e.g., Oracle, PostgreSQL, FireBird...) and others implement it (e.g. Microsoft's SQL Server). Therefore, current application server implementations are incorrect when used with databases providing snapshot isolation.

Snapshot isolation provides a similar level of isolation as serializability (it passes the tests for serializability of standard benchmarks such as the ones from TPC [Tra]). Snapshot isolation is usually implemented via a multi-version mechanism in which transactions see a snapshot of the database as it was when the transaction started [BBG+95]. Therefore, readers and writers do not interfere. In contrast, when implementing serializability, read-write conflicts lead to blocking or aborts, reducing the potential concurrency and the performance of the system.

J(2)EE implementations provide caching for entity beans (EBs) in the EJB container as follows. An EB represents a cached tuple of the database. The entity bean representing tuple x is denoted as EBX. When accessing an EB, if it is not in memory the corresponding tuple is read from the database. If the EB is updated, the associated tuple will be updated at the database when the corresponding transaction commits. The EB is then cached in memory so that it can be directly accessed by further transactions. Access to EBs is typically controlled via locking in order to provide serializability. However, this can lead to executions that neither provide serializability nor snapshot isolation when the DBMS uses snapshot isolation.

Let's look at the example shown in Figure 4.2. Assume two transactions $T1$ and $T2$ start concurrently at the application server. $T1$ wants to write x and y. For that, the application server reads the values of x and y into EBX and EBY, respectively, $T1$ gets locks and updates both beans. Concurrent transaction $T2$ reads z into EBZ, and then wants to read y but is blocked because $T1$ has a lock on EBY. Now $T1$ commits. The new values for x and y are written to the database and the locks are released. Both EBX and EBY remain cached.

$T2$ receives the lock on EBY and reads the current value of EBY, namely the value written by $T1$. Now assume the cache replacement policy evicts EBX from the cache. Later, $T2$ wants to read x. x is re-read into a new incarnation of EBX. However, since the database uses snapshot isolation and $T2$ is concurrent to $T1$ in the database, the old value of x is read. Therefore, $T2$ reads for y the value written by $T1$ but for x a previous value. This does neither conform to snapshot isolation nor to serializability.

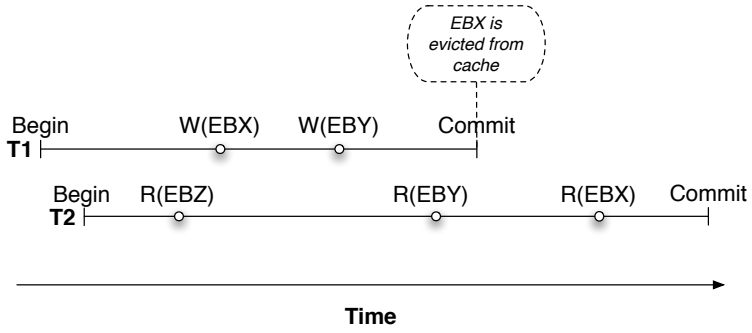


Figure 4.2: Regular caching anomaly with SI

In order to avoid the anomalies of J(2)EE caching when used with a snapshot isolation database, here it is proposed a multi-version cache for EBs that enforces snapshot isolation. For each EB, instead of keeping a single copy in the cache, a list of potentially more than one version is cached. Each bean version EBX_i of data item x is tagged with the commit timestamp i of the transaction that created (and committed) this version. Additionally, the multi-version cache is replicated at several application servers in order to provide scalability, and availability. The semantics of the replicated multi-version cache is as if there was a single multi-version cache providing snapshot isolation (one-copy snapshot isolation).

The communication among the replicas of the cluster is provided by a *group communication system* (GCS). The protocol requires from the GCS total order reliable multicast [CKV01]. This means that all the messages are delivered to the group members in the same order in which they were sent. Moreover, it is assumed that multicast messages are delivered also to the sender of the message (self-delivery). GCS also provides group membership facilities, that are used to control the replicas connected to the cluster at a certain point in time (view). The protocol assumes, primary component membership. In primary component membership, the views installed by all members are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one member that survives from the one view to the next one. Finally, *strong virtual synchrony* is also required. Strong virtual synchrony guarantees that the relative order of delivering membership changes (views) and multicast messages is the same in all replicas.

The process of rejoining a failed or a fresh replica to a cluster is called *recovery*. In principle, it seems easy to recover a replica: just as simple as starting a new replica and joining it to the cluster's infrastructure. However, this process is not so simple if the clients can update data concurrently in the cluster. In this case, the data must be kept consistent in all the replicas. So, when a replica is added to a cluster, the recovery process must guarantee that the data of the new replica is also consistent with the data hold on the other replicas. Moreover, the recovery process should not conflict with the normal operation of the applications deployed

in the cluster. Clients should be able to perform requests and receive responses from the applications whilst the recovery process is being done, what is called *online recovery*. There are two roles for the replicas in the online recovery process: *recovering* and *recoverer*. On the one hand, the recovering is the replica that joins to the cluster in order to be recovered; on the other hand, the recoverer replicas take care of transferring their current consistent state to the recovering replica.

Combining all these concepts and elements (update everywhere, vertical replication, cache, snapshot isolation, online recovery...), it is possible to build a high performance cluster that also provides high availability and consistency for the data of multi-tier applications.

4.3. System Model

The replication and recovery approach for the protocol is based on the update everywhere approach. Each replica uses a vertical replication scheme. Figure 4.3 shows the configuration.

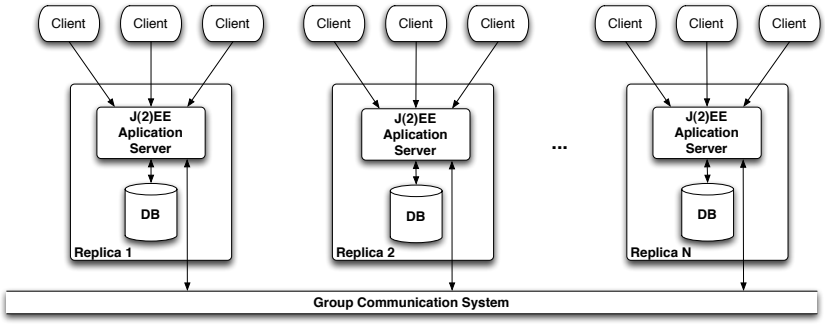


Figure 4.3: Replication model

In the update everywhere approach, clients may perform update operations in any of the replicas of the cluster. In the vertical replication schema deployed, a J(2)EE application server and a database management system (DBMS) are collocated in the same node. The set of all replicas configured in such manner is called a *high performance cluster*.

The system model only considers crash failures of the replicas. A replica is crashes if either the application server or the database crashes. This is natural since they are collocated. If the cluster contains N replicas ($N > 1$), it supports $N - 1$ failures. The failure of the replicas is transparently masked to the clients. If the client is connected to a replica that crashes, when the failover is triggered, the client automatically connects to any other replica in the cluster and performs a re-invocation.

The clients of J(2)EE applications invoke transactional methods on session beans (SBs) in any of the replicas of the cluster. The particular replica where the client is connected to is in charge of managing the required transaction for the client request. Then, the SBs bean may invoke other session or entity beans in the same transactional invocation. The nested transactions that may be triggered when invoking other components are not considered by the protocol. Thus, there is a one-to-one relationship between client requests and transactions.

The changes in the stateful components produced in each replica must be sent and applied in all the replicas of the cluster. The replication protocol relies on the features provided by

a group communication system (GCS) for implementing the membership and communication among the replicas. The GCS running in each application server requires strong virtual synchrony and must be configured to use primary component membership. The messages are delivered to the replicas using uniform reliable multicast with total order guarantees.

4.4. High Available and Scalable Replication Protocol

In this section is described the multi-version cache replication protocol. An overview of the protocol is presented first. Then, the protocol is discussed in detail.

The protocol works as follows; When a request is submitted to a replica, a transaction is started at both, the application server (AS) and the database (DB). The transaction might read data that is already cached at the AS or that has to be read from the DB. The cache protocol makes sure that the correct version is read according to snapshot isolation. If the transaction is read-only, it simply commits locally and the result is returned to the client. If the transaction updates a data item x, a new private version of the corresponding entity bean EBX is created. At commit time, the replication protocol multicasts all EB versions created by the transaction (e.g. its *writeset*) using a total order multicast provided by the group communication system. That is, although different replicas might multicast at the same time, all replicas receive all writesets in the same order. At each replica, the replication protocol now validates incoming writesets in the same order. If validation determines that a concurrent transaction that already validated had an overlapping writeset, validation fails and the transaction is aborted. If validation succeeds, the replica assigns a commit timestamp, tags the EB versions of the transaction with the commit timestamp and adds them to the cache. Then the transaction commits at AS and DB. When the transaction commits at the local replica, the result is returned to the client. Each replica validates the same set of update transactions in the same order, and decides on the same outcome for each individual transaction. Thus, each committed transaction has the same commit timestamp at each replica.

4.4.1. Protocol Details

This section discusses in detail how transactions are managed by the protocol. When a transaction is submitted to a replica R, the replica starts a local transaction T at the local AS and a transaction t at the local DB. The correlation between AS transactions and DB transactions is stored in a table (Figure4.4 line 6). Each transaction T at the AS will be associated with a *start timestamp* $ST(T)$ when it starts (line 3), and a *commit timestamp* $CT(T)$ at commit time (line 54) which is assigned from a counter that is increased every time a transaction commits. The start timestamp $ST(T)$ of a transaction T is the highest commit timestamp $CT(T')$, and indicates that T should read the committed state that existed just after the commit of T'. We assume that the initial start timestamp is 0 at all replicas. Each bean version $EBX_{CT(T)}$ of a data item x is tagged with the commit timestamp $CT(T)$ of the transaction T that updated (and committed) this version. When a transaction T reads a data item x it has written before, it reads its own version (lines 10-11). Otherwise, it first looks for EBX in the cache. It reads the version EBX_i such that $i \leq ST(T) \wedge \nexists EBX_j : i < j \leq ST(T)$, e.g. it reads the last committed version as of the time it starts (lines 12-13).

If no appropriate bean is cached in memory, the transaction reads x from the DB and the corresponding EBX version is created (lines 15-18). Since a transaction is started at the DB when a transaction starts at the AS, and the DB provides snapshot isolation, the DB will return the correct version for x. This process guarantees that each transaction observes a snapshot as of the start of the transaction and therefore it does not violate snapshot isolation. Since the DB does not show the versions associated to tuples, when a version of data item x is read from the DB the corresponding tag of the EBX version is unknown. Thus, the bean is tagged with -1.

In order to guarantee that transactions always read the appropriate bean version, the cache guarantees that for each data item x the following holds: (i) If both T and T' updated x, and $CT(T') > CT(T)$ and the version $EBX_{CT(T)}$ is cached, then $EBX_{CT(T')}$ is also cached. That is, if the cache contains a certain version of a bean, then it also contains all later versions of this bean; (ii) If there exists a version $EBX_{CT(T)}$ and there exists an active local transaction T' that is concurrent to T, e.g., $ST(T') > CT(T)$, then $EBX_{CT(T)}$ is cached. That is, a version is cached at least as long as there exists a concurrent local transaction that has not yet terminated. Having all these versions cached is important for reads.

Note that our approach requires the DB to provide snapshot isolation. This is needed because the cache cannot keep the entire database, e.g., all versions of all tuples. To show that snapshot isolation is needed at the DB level, assume the DB uses the isolation level read committed (or serializability via locking). Assume further that a transaction T_i reads and modifies x while a concurrent transaction T_j updates y and commits. Assume now further that T_i wants to read y and, due to lack of memory, the version of EBY that T_i needs to read, was evicted from the cache. Hence, it has to read it from the DB. Assuming transactions run at the DB with read committed (or serializability) isolation level, T_i reads the value of y committed by T_j. That is, it will not read the value of EBY at the time T_i started. Thus, the AS cannot provide by itself the snapshot isolation level.

In snapshot isolation, when two concurrent transactions update the same data item, only one may commit, the other has to abort. In order to detect such conflicts early both, locking and version checking, have been used. When a transaction T_i wants to update a data item x, it has to first acquire a write lock on EBX (line 25). Write locks guarantee that at most one transaction updates data item x at any time. If another transaction holds a lock on EBX, T has to wait until the lock is released which is done at transaction abort (line 45) or commit (line 73). Lock requests are inserted into a FIFO wait-queue, e.g., when a transaction releases a lock the first in the wait queue receives it. Once a transaction T has a lock, a version check on the version EBX_j with the highest version number in the cache is performed. If $j > ST(T)$, then this version was created by a concurrent, already committed transaction, and T must abort (lines 26-27). If no such version exists, T can perform the update, e.g., create its own version and add it to its writeset ($WS(T)$) (line 29-31). For now, this version is only seen by T itself. This guarantees that a transaction observes its own updates (lines 10-11) and prevents other transactions from observing uncommitted changes.

When a transaction wants to commit, if the transaction was read-only it is simply committed at the DB (lines 36-37). Otherwise, the writeset is multicast to all replicas using a total order multicast (line 39). All replicas in the multicast group (including senders) receive all messages in the same order. When a replica processes such a message (line 50), the corresponding transaction performs a final validation (line 53). This will help to find conflicts among transactions that executed at different replicas. Since all transactions perform deter-

```
Data:
timestamp = 0;
cache = ∅ ;
committedTx = ∅;
transactionTable = ∅;
mutex;
oldestActiveTx = array[1..NumberReplicas] of Int =
0;
1  Begin(T)
2  |  set mutex ;
3  |  ST(T) = timestamp;
4  |  WS(T) = ∅;
5  |  t = begin transaction in the DB;
6  |  store(transactionTable, T, t);
7  |  release mutex ;
8  end
9  Read(T, EBX)
10 |  if EBX_Tprivate ∈ WS(T) then
11 |  |  return EBX_Tprivate ;
12 |  else if ∃EBX_i ∈ cache : i = max(j) | EBX_j ∈
   |  cache ∧ j < ST(T) then
13 |  |  return EBX_i ;
14 |  else
15 |  |  t = getTx(transactionTable, T);
16 |  |  EBX_-1 = read(t,EBX) from the DB;
17 |  |  cache = cache ∪ {EBX_-1};
18 |  |  return EBX_-1;
19 |  end
20 end
21 Write(T, EBX, value)
22 |  if ∃EBX_Tprivate ∈ WS(T) then
23 |  |  write(EBX_Tprivate, value);
24 |  else
25 |  |  acquire lock on EBX for T;
26 |  |  if ∃EBX_i ∈ cache | i > ST(T) then
27 |  |  |  abort(T);
28 |  |  else
29 |  |  |  create(EBX_Tprivate);
30 |  |  |  WS(T) = WS(T) ∪ {EBX_Tprivate};
31 |  |  |  write(EBX_Tprivate, value);
32 |  |  end
33 |  end
34 end
35 Commit(T)
36 |  if WS(T) == ∅ then
37 |  |  Commit (getTx(transactionTable, T)) in DB;
38 |  else
39 |  |  multicast(WS(T), T,
   |  |  minLocalTx(transactionTable));
40 |  end
41 end
42 Abort(T)
43 |  ∀ EBX_Tprivate ∈ WS(T) do
44 |  |  delete(EBX_Tprivate);
45 |  |  release lock on EBX;
46 |  end
47 |  abort getTx(T) in DB;
48 |  delete(transactionTable, T);
49 end
50 Upon delivery of (WS(T), T,
   oldestLocalActiveTx)
51 |  set mutex;
52 |  oldestActiveTx[Sender(T)] =
   |  oldestLocalActiveTx;
53 |  if ∄T_K ∈ committedTx : ST(T) < CT(T_k)∧
   |  WS(T) ∩ WS(T_K) ≠ ∅ then
54 |  |  CT(T) = ++timestamp;
55 |  |  if local(T) then
56 |  |  |  ∀EBX_Tprivate ∈ WS(T) do
57 |  |  |  |  replace tag Tprivate with tag CT(T);
58 |  |  |  |  cache = cache ∪ {EBX_CT(T)};
59 |  |  |  end
60 |  |  else
61 |  |  |  t = begin transaction in the DB;
62 |  |  |  store(transactionTable, T, t);
63 |  |  |  ∀EBX_Tprivate ∈ WS(T) do
64 |  |  |  |  if ∃ local transaction LT that has lock
   |  |  |  |  on EBX then
65 |  |  |  |  |  abort(LT)
66 |  |  |  |  end
67 |  |  |  |  acquire lock on EBX for T (put lock
   |  |  |  |  request at begin of wait queue);
68 |  |  |  |  replace tag Tprivate with tag CT(T);
69 |  |  |  |  cache = cache ∪ {EBX_CT(T)};
70 |  |  |  end
71 |  |  end
72 |  |  commit (getTx(transactionTable, T)) in the
   |  |  DB;
73 |  |  ∀EBX ∈ WS(T) release lock on EBX;
74 |  |  committedTx = committedTx ∪ {T};
75 |  |  delete(transactionTable, T));
76 |  release mutex;
77 end
```

Figure 4.4: Replicated cache protocol for replica R

ministic validation in the same order, all replicas decide on the same outcome. A transaction passes validation, if there is no transaction in the system that is concurrent, already committed and has overlapping changes. In order to contrast this information, a list structure called *committedTx* is used. Each time a transaction passes the validation and commits, it is added

to this list (line 74).

When a transaction T passes validation, it receives a commit timestamp (line 54). At the local replica, the private versions are tagged with the commit timestamp and added to the cache (lines 55-59). T and the corresponding DB transaction t commit and the locks are released (lines 72-73). Note that committing the DB transaction automatically propagates the changes to the DB. The protocol keeps track of all committed transactions (line 74) for validation purposes. At a remote replica, a DB transaction is started for T (lines 61-62). T first gets the locks on the data items. If a local transaction T' has a lock on one of the data items it has to abort because it is concurrent to T, has updated the same data item and is not yet validated and committed (lines 64-65). T has to be the first to get the lock (line 67). Then, the versions sent in the message are tagged with the commit timestamp and added to the cache (lines 68-69). From there, the transaction commits as in the local replica (lines 72-75).

If a transaction T does not pass validation, nothing has to be done. At remote replicas, the message can simply be discarded since nothing has yet been done on behalf of T. At the local replica, T can only fail validation if a conflicting remote transaction T' was received between sending and receiving T. In this case, however, as described above T' found the lock held by T and T was already forced to abort.

Note that messages are processed serially, that is, one after the other, in order to guarantee that validation and commit order are the same at all replicas. Furthermore, starting transactions have to be coordinated with committing transactions in order to guarantee that transactions see, in fact, the correct snapshot. Therefore, an appropriate mutex is set (lines 2,7, 51 and 76).

4.4.2. Examples

The execution is illustrated along two examples. Figure 4.5 shows an example of the evolution of the cache on a single replica (ignoring the replication part). It is assumed that the cache is empty and the commit counter is at value 10. Transactions $T1$ and $T2$ obtain the same start timestamp (10) and each creates a corresponding DB transaction. Then, $T2$ reads x. Since no bean version exists in the cache, the data item is read from the DB and a version EBX_{-1} is created. The value of x is a. Now $T1$ reads x and y. Since EBX_{-1} is cached and $-1 \leq 10$, $T2$ reads EBX_{-1}. Furthermore y is read from the DB and stored in EBY_{-1}. Its current value is b. Now $T1$ updates EBX to the value c and EBY to the value d. For that, it creates private versions of EBX and EBY. Finally, it requests the commit. It receives commit timestamp $CT(T1) = 11$, the versions are tagged with this timestamp and added to the cache. The corresponding DB transaction commits meaning that the changes are transferred to the DB. Since the DB implements SI, new versions for both x and y are created also in the DB. When $T2$ now reads y, it does not read EBY_{11}, since $11 > ST(T2)$. Instead, it reads EBY_{-1} that is, the old value b of y. Since $T2$ is read-only, it simply commits in the DB and no commit timestamp is assigned.

Figure 4.6 shows an example with two replicas –$R1$ and $R2$– are shown. It is assumed that the commit counter at each replica is 10 when transaction $T1$ starts at $R1$ and $T2$ at $R2$. Both receive start timestamp 10. Now assume both read data item x, reading it from the local DB and loading it into EBX_{-1}. The current value is a. Now both transactions update EBX. Since they run in different replicas, both acquire the lock, and create their own private

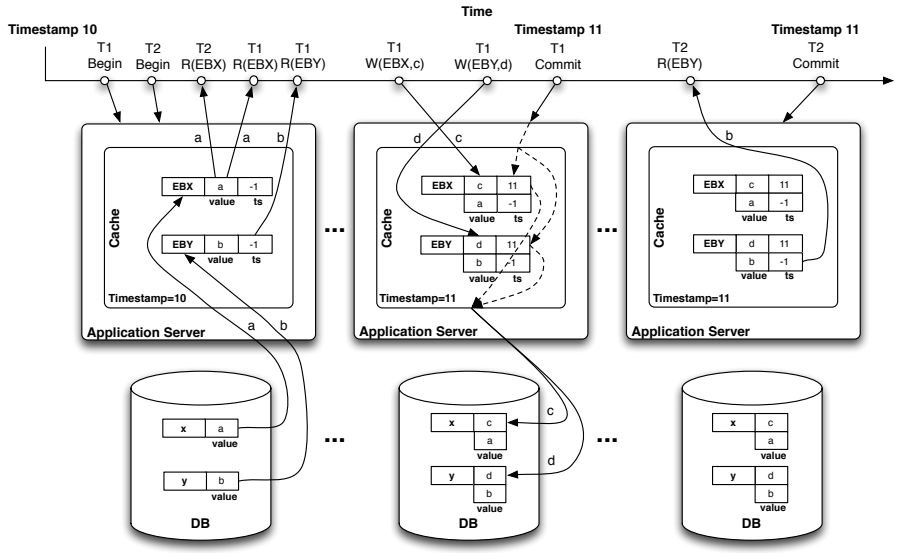


Figure 4.5: Evolution of the cache in a single replica

EBX versions. $T1$ sets the new value b, $T2$ sets the new value c. When $T1$ and $T2$ finish at their local replicas, their changes are multicast. Let us assume the total order is $T1, T2$ and there is no other concurrent conflicting transaction. Let's first have a look at $R1$. When $T1$ is delivered at $R1$, its validation succeeds. $T1$ is local at $R1$. It receives the $CT(T1) = 11$ and its version is tagged ($EBX_{11} = b$) and added to the cache. $T1$ commits at the DB. When now $T2$ is delivered at $R1$ validation fails since $T1$ is concurrent ($CT(T1) > ST(T2)$), conflicts, and has already committed. Therefore, nothing is done with $T2$ at $R1$. At replica $R2$ transactions are validated in the same order. $T1$'s validation succeeds. $T1$ is a remote transaction at $R2$ and has to acquire the locks. However, $T2$ has a lock on EBX. $T2$ is local and has not yet validated. Therefore, it is aborted, its private version discarded and its lock released. $EBX_{11} = b$ is created and added to the cache. The value is propagated to the DB and the transaction committed. When later $T2$ is delivered at $R2$, validation fails. The transaction has already aborted, and nothing has to be done. Therefore, the two replicas commit the same transactions and keep the same values (with the same version tag) in both the cache and the DB.

4.4.3. Dealing with Creation and Deletions of EBs

Creation and deletion of EBs is also handled by the protocol (not shown in Figure 4.4). When a new EB is created (no corresponding data item exists in the DB), a private version is created for the transaction and there is no other version available. A lock is also set on the EB to prevent concurrent creations of the same EB (with the same primary key). When the transaction commits, the version becomes available for transactions that started after the creating transaction committed and the corresponding tuple is inserted in the DB.

Deletions create a tombstone version of the EB. The tombstone is also a private version of the transaction until commitment. If the transaction tries to access the EB, it will not

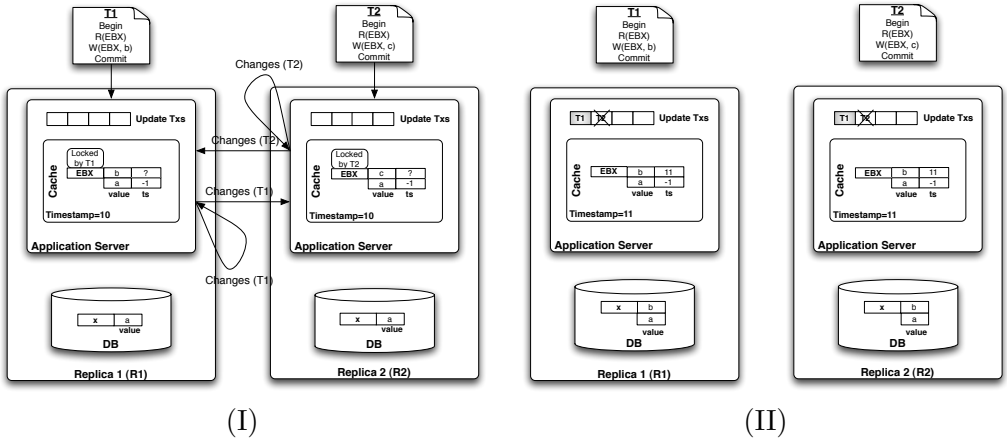


Figure 4.6: Two concurrent conflicting transactions

find it, since the protocol will find the tombstone and recognize the EB as deleted. When the transaction commits, the tombstone version will become public. Even after transaction commit previous versions of the EB cannot be removed, since there might be active transactions associated to older snapshots (all transactions that started before the one that deleted the EB committed), that may read the older EB version.

4.4.4. Garbage Collection

Since EB versions are kept in memory (in the cache), they should be removed to free space in the cache when they are not needed. For this purpose, there is a *garbage collection mechanism* that discards unused EB versions. The pseudocode for the garbage collection is shown in Figure 4.7.

```
78 garbageCollection()
79   oldestTx = min(oldestActiveTx);
80   ∀T ∈ committedTx do
81     if CT(T) < ST(oldestTx) then
82       committedTx = committedTx - {T} ;
83     end
84   end
85   oldestLocalTx = oldestActiveTx[R];
86   ∀EBX ∈ cache do
87     if ∃EBX_i ∈ cache ∧ i ≠ −1 ∧ i <
       ST(oldestLocalTx) then
88       cache = cache - EBX_i;
89       if EBX_−1 ∈ cache then cache =
         cache-EBX_−1;
90     end
91   end
92 end
```

Figure 4.7: Garbage collection

Each replica removes versions of components that are older than the oldest start timestamp among local active transactions (lines 85-88). If a version of an EB, EB_i, is not needed (there is no local active transaction with start timestamp smaller than i), then version EB_{-1} is also not needed, since EB_{-1} is older (line 89). Moreover, if EB_{-1} is not evicted from the cache and a new transaction T ($ST(T) > i$) reads EB, it would read EB_{-1}, which is incorrect, since it should read a later committed version (EB_i or even a later version, since $ST(T) > i$).

Uncommitted updated EBs are pinned in the cache. If the cache gets full with pinned EBs, the J(2)EE application server writes locked EBs from the cache to a local disk repository (not the database) by means of a standard hibernation mechanism. Thanks to this, our versioning is not affected by the eviction policy of the cache. When a hibernated EB is going to be accessed the AS brings the hibernated EB to memory including all EB versions and their tags. Note that updated EBs whose changes have been committed will be evicted from the cache according to the cache policy.

Writesets ($committedTx$) are also garbage collected. Since they are used for validation, a writeset can only be garbage collected when there are no more active concurrent transactions in the system (lines 79-82).

4.4.5. Session Replication

Stateful session beans (SFSBs) keep conversational state from a client and their replication is not required to provide consistency and availability of EBs. However, if they are not replicated, a failure of a replica will cause the loss of the conversational state kept in the SFSB corresponding to all previously run transactions by the client at that replica. The conversation could not be resumed after the failover, what results in loss of session availability. For this reason, the replication protocol also replicates the state of SFSBs after each method invocation.

Since SFSBs are only replicated to increase availability when a failure occurs one could consider to replicate them to one or two replicas at the most (what is called buddy replication).

An SFSB is checkpointed to the other application server replica/s before returning the result to the client. Additionally, the response to be sent to the client is also checkpointed. During failover, another replica will take over and the checkpointed SFSB will become active and process the next client requests. Exactly once semantics is guaranteed by means of resubmission of the last client outstanding request and duplicate removal. It should be noticed that exactly once semantics can be supported thanks to checkpointing the request result. Otherwise, the following scenario might happen if the request was processed at the failed replica, but the request result did not reach the client. When failing over, the client will resubmit the outstanding request to the replica taking over. The request will be recognized as a duplicate and discarded. However, the client does not have the request result. Additionally, since the request was already processed, it will no be processed again by the replica that took over. By checkpointing the request result, the replica taking over can return the request result for the resubmitted request. The previous chapter focuses in deep on this topic.

4.4.6. Failure Handling

Clients connect to the application server through stubs that are obtained from the application server through JNDI (Java Naming and Directory Interface). Since stubs are generated

by the application server, the necessary replication logic can be added in a fully transparent way to clients. The stubs have been extended to be able to perform replica discovery and load-balancing, relying on IP-multicast. When a client wants to connect, the stub IP-multicasts a message to an IP-multicast address associated to the application server cluster. Clients are identified by a unique client identifier. When the replicas in the cluster receive a connection request, one of them (depending on the client identifier) returns to the stub a list of available replicas (their IPs) as well as an indication of their current load. Replicas multicast information about their load periodically (e.g., piggybacked on the writeset message). The stub then selects a replica randomly with a selection probability inversely proportional to the load of the replicas to attain load balancing. The stub connects to the selected replica and sends all client requests to this replica (sticky client). Each request receives a unique number (a counter kept at the stub that is incremented after each successful request).

The AS replicas build a group using the group communication service. The group communication system provides the notion of view (currently connected cluster members). Whenever a member fails, the available members are informed via a view change message. The group communication system provides strong virtual synchrony that guarantees that the relative order of delivering view changes and multicast messages is the same at all replicas. The total order multicast used for the writeset messages in the replication protocol also provides reliable delivery guaranteeing that all available replicas receive the same set of messages [CKV01]. Furthermore, the writeset also contains the client identifier, request identifier plus the response that is going to be returned to the client. The remote replicas store for each client the latest request identifier, the outcome of the transaction (commit/abort), and in case of commit, the response.

Let us now consider the failover logic at the application server side. Each replica consists of a pair of AS and DB. If any of them fails or the node in which they are collocated fails, the replica is considered as failed. If only the AS or the DB fails, the other component automatically shuts down. Only crash failures are assumed.

At the client side the failure will be detected when the stub times out waiting for the response to an outstanding request. The stub will reconnect to a new replica and resubmit that request. Notice that container managed transactions are being considered, where each client request will be automatically bracketed as a transaction. Thus, there is a one-to-one relationship between requests and transactions. A failure can now occur at two logical points. (1) The replica failed before multicasting the writeset related to the request to the other replicas. (2) The replica failed after multicasting the writeset.

If there have been previous interactions with that client, the new replica to which the stub connects to will have the last state of the stateful session bean (SFSB) associated to the client and processes the resubmitted client request in the following way. In case (1), the new replica does not yet have any information about this request and thus, will process it as a new request. In case (2) it has already stored the request identifier and the outcome of the corresponding transaction. It recognizes the resubmitted request as a duplicate for which it has already the outcome stored. If the outcome was commit, it returns the response to the client. Otherwise, it returns an exception to the client notifying that the transaction was aborted since snapshot isolation could not be guaranteed (the failed replica would have done the same if it had not failed).

4.5. Online Recovery Protocol

This section describes a recovery protocol that complements the multi-version cache replication protocol in order to add new replicas to a high performance cluster or to recover the failed ones. First of all, an overview of the protocol is presented.

The application servers of each replica must hold the writesets of committed transactions in a log file in order to be able to perform the recovery process. When transferring the state to the replica two possible scenarios may arise. This depends on the amount of writesets to send to the recovering replica from the application server logs of the current online replicas (recoverers). If the number of writesets is not too high, the recovery process is performed using only the application server logs. Otherwise, it is worth to transfer first an snapshot of the database and then complete the process using the application server logs.

Let's imagine the first of the previous two scenarios. The recovery protocol works as follows; when a replica needs to be recovered (either a failed or a fresh replica), all the online replicas collaborate to transfer the state to it acting as recoverers. The recovering replica informs the online replicas about the number of the last writeset that it committed. Then, each online replica sends to the recovering consecutive chunks of writesets from the application server log starting from the last writeset number received. The recovering collects the writesets in the correct order and commits the writesets in the database. When the last chunk of writesets is about to be sent, the recovering starts caching in FIFO order the new writesets of transactions that are going to be validated by the other replicas. Upon receiving and applying the last chunk of writesets coming from the recoverers, these cached writesets allow the recovering to complete the state synchronization; the recovering extracts the cached writesets and uses the validation process of the replication protocol in order to commit or not the transactions.

4.5.1. Protocol Details

This section describes the online recovery protocol in detail. The protocol is explained from two different points of view. The first one is the point of view of the recovering replica, that is, the replica that joins the cluster in order to be recovered. The second is the point of view of the recoverer replicas, that are the online replicas that transfer their current consistent state to the recovering replica. Among the recoverers, there is a special replica called *coordinator* that is in charge of driving the recovery process. The variable *role* defines the function of each replica in the protocol (recovering, recoverer or coordinator). Initially, the recoverers are the replicas that are online in the cluster. The recovery protocol uses the same communication channel of the group communication system as the replication protocol in order to multicast the messages with total order guarantees to all the replicas in the cluster.

Each replica R_i acting as a recoverer collaborates in the recovery process. The required information for the recovery is in the writesets of all transactions that each replica has committed. The replication protocol introduced previously, keeps track of the writesets associated to committed transactions in a list structure called *committedTx*. When garbage collection is done in the replication protocol, each replica removes from committedTx the writesets (WSs) of committed transactions that are older than the oldest start timestamp common to all the replicas in the cluster (These are the writesets of committed transactions that are not concurrent with the current running transactions. See Figure 4.7, lines 79-84). These writesets removed are placed in an *application server log file*. Thus, the required writesets to recover a

replica are contained in the application server log file and in the committedTx structure hold in memory.

When transferring application server state from the recoverers to the recovering replica, the transferred state must be consistent with the underlying database state. Therefore, a mechanism to correlate the state hold at the application server level with regard to the persistent state stored in the database is also required within the application server. In the replication protocol, the current database state is represented in memory at the application server by the commit timestamp (CT) of the latest committed transaction in the replica (See Figure 4.4 line 54). This information must be resilient to failures of the replica, so the last CT is stored in a new table named *DBState* in the application's database. Thus, when a transaction is about to be committed in all the replicas, an update of the CT in the DBState table must be executed as part of the same transaction.

The pseudocode of the protocol is shown in Figure 4.8. The recovery process can be done sending only writesets contained in the application server log or sending a snapshot of the database at a certain point in time and the required writesets from the application server log. The following paragraphs explains the process for the first case. Section 4.5.3 explain the protocol when a database snapshot must be sent.

The recovery protocol is triggered by the recovering node (lines 1-5). The recovering multicast in total order a message (*StartRecovery*) to the recoverers including the identifier of the last transaction that committed before its crash (*timestamp*). This information is obtained by the *getLastDBTimestamp()* function from the DBState table. When the recovering replica becomes a new member of the cluster, a view change occurs. Let's call V_n the new view installed. A particular replica R_i among the recoverer replicas is elected as coordinator, that is, $R_i \in (V_{n-1} \cap V_n)$ (lines 38-42).

When the coordinator receives the message containing the timestamp from the recovering replica, it decides through the *isTooOld(timestamp)* method either to send the whole database to the recovering or just send the writesets in the application server log (lines 43-57). The isTooOld() function compares the received timestamp with the current timestamp at the replica and analyzes the size of the writesets to send. Let us assume that it is not necessary to send the whole database. Thus, in order to recover the recovering replica, the recoverers (the coordinator is also a recoverer) start sending chunks of the log to the recovering replica (lines 49-53). The *getNextChunk()* function returns a chunk of writesets to each recoverer. The number of writesets in each chunk is specified by *CHUNK_SIZE*. In order to not to send overlapping writesets, getNextChunk() uses a deterministic approach to select the chunk from the application server log to send based on the number of recoverer replicas in the cluster.

Upon receiving application server log chunks, the recovering replica stores the chunks in order in the application server log (line 16). Then, the recovering replica extracts one by one the writesets from the log (lines 17-23). For each writeset, it creates a database transaction, extracts the database updates from the writeset and applies them, and finally commits the transaction (lines 18-20). The committedTx structure and the timestamp are also updated (lines 21-22) in order to allow the later validation of the writesets of transactions that are being executed at the same time by other replicas.

A synchronization point is needed to stop sending the application server log to the recovering replica. This is achieved by means of the following mechanism. When the amount of writesets in a chunk is lower than a certain threshold (e.g. CHUNK_SIZE), the recoverers stop sending messages with log chunks (line 50). Then, the coordinator multicast using total order

```
   Data:
   CHUNK_SIZE = K;
   log = ∅;
   role = ONLINE;
   mutex;
   timestamp = X;
   committedTx = ∅;
   // Recovering Replica
 1 Upon a replica joins the cluster:
 2 |  role = RECOVERING;
 3 |  timestamp = getLastDBTimestamp();
 4 |  multicast(StartRecoveryMsg(timestamp));
 5 end
 6 Upon delivering DBBackupMsg(dbBackup):
 7 |  install(dbBackup);
 8 |  startDBMS();
 9 |  timestamp = getLastDBTimestamp();
10 |  multicast(StartRecoveryMsg(timestamp));
11 end
12 Upon delivering PreOnlineMsg():
13 |  start caching transactions;
14 end
15 Upon delivering LogMsg(logChunk):
16 |  log.add(logChunk);
17 |  for (log.currentElem() to log.lastElem()) do
18 |  |  begin transaction in the DB;
19 |  |  WS(T) = log.nextElem();
20 |  |  commit WS(T) in the DB;
21 |  |  committedTx = committedTx ∪ {T};
22 |  |  timestamp++;
23 |  end
24 end
25 Upon delivering
   LastChunkMsg(lastLogChunk):
26 |  log.add(lastLogChunk);
27 |  for (log.currentElem() to log.lastElem()) do
28 |  |  begin transaction in the DB;
29 |  |  WS(T) = log.nextElem();
30 |  |  commit WS(T) in the DB;
31 |  |  committedTx = committedTx ∪ {T};
32 |  |  timestamp++;
33 |  end
34 |  status = ONLINE;
35 |  start validating cached transactions;
36 end
   // Coordinator/Recoverer Replica
37 Upon delivering
   StartRecoveryMsg(timestamp):
38 |  if (iAmCoordinator(R)) then
39 |  |  role = COORDINATOR;
40 |  else
41 |  |  role = RECOVERER;
42 |  end
43 |  if (isTooOld(timestamp)) then
44 |  |  if (role == COORDINATOR) then
45 |  |  |  dbBackup = createDBBackup();
46 |  |  |  unicast(recovering,
   |  |  |  DBBackupMsg(dbBackup));
47 |  |  end
48 |  else
49 |  |  nextChunk = getNextChunk();
50 |  |  while
   |  |  (nextChunk.size() == CHUNK_SIZE) do
51 |  |  |  unicast(recovering, LogMsg(nextChunk));
52 |  |  |  nextChunk = getNextChunk();
53 |  |  end
54 |  |  if (status == COORDINATOR) then
55 |  |  |  multicast(PreOnlineMsg());
56 |  |  end
57 |  end
58 end
59 Upon delivering PreOnlineMsg():
60 |  set mutex;
   |  // Read the current timestamp at Replica R
61 |  lastWSToSend = timestamp;
62 |  release mutex;
63 |  nextChunk = getNextChunk();
64 |  while ((nextChunk.size() >
   |  0) ∧ (getMaxTimestamp(nextChunk) <
   |  lastWSToSend)) do
65 |  |  unicast(recovering, LogMsg(nextChunk));
66 |  |  nextChunk = getNextChunk();
67 |  end
   |  // Send the last chunk trimmed to the last
   |     WS to send
68 |  if (lastWSToSend ∈ nextChunk)) then
69 |  |  lastChunk = nextChunk.trim(lastWSToSend);
70 |  |  unicast(recovering,
   |  |  LastChunkMsg(lastChunk));
71 |  end
72 |  status = ONLINE;
73 end
```

Figure 4.8: Recovery protocol for replica R

a *PreOnline* message to indicate that the recovery is finishing (line 55). When the PreOnline message is delivered in the recovering replica, it starts caching all the new incoming writesets that are being processed by the other replicas. On the other hand, when the recoverers receive the PreOnline message, they obtain the timestamp of last committed transaction at that point (line 61). As the recoverer replicas continue processing client requests, the timestamp must be obtained in mutual exclusion to not interfere with the replication protocol. This timestamp represents the last writeset from the application server log that must be send to the recovering

replica (*lastWSToSend*).

After the synchronization point has been established, the recoverer replicas continue sending to the recovering all the missing writesets between the sending and the delivery of the PreOnline message (lines 63-67). Finally, the recoverer replica that contains the last required chunk of the application log sends it to the recovering replica using the *LastChunkMsg* message (lines 68-71). From this point on, the recovery process has finished for the replicas acting as recoverers (line 72).

Upon adding the last log chunk contained in the LastChunk message to the log, the recovering replica commits the writesets contained in it into the database (lines 25-36). When all of the writesets of the log have been committed into the database and the EJB components have been recreated or updated, the recovering replica has been recovered. Therefore, it can start validating the cached transactions from the other replicas using the replication protocol. Finally, when the cached transactions have been validated, the recovering replica is ready to receive new requests from clients.

4.5.2. Example

Figure 4.9 shows an example of a recovery scenario. The scenario describes how a replica that crashed after committing the transaction with Timestamp = 2 is recovered. When the replica re-joins the cluster as a recovering replica, the last committed transaction in the other replicas has a Timestamp = 10. In the top of the figure are shown the application server logs of each online replica participating as a recoverer. It is assumed a CHUNK_SIZE = 2 (Depicted in gray).

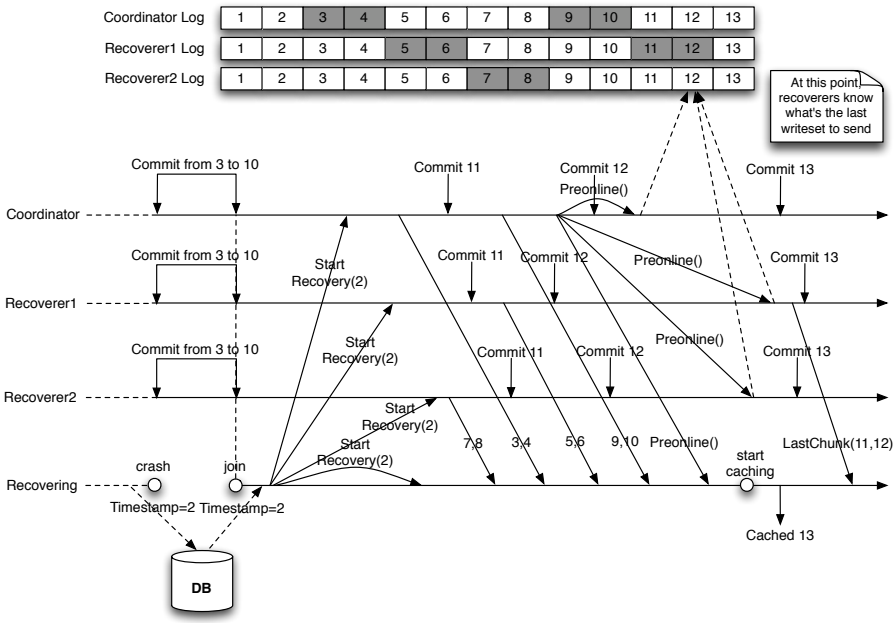


Figure 4.9: Parallel recovery process

First of all, the recovering multicasts to all online nodes the StartRecovery message con-

taining the timestamp of its last committed transaction before crashing, in this case is equal to 2.

After having received the StartRecovery message, all online nodes behave as recoverer replicas. The coordinator is shown at the top. Then, the coordinator and the recoverers start to send all the missing writesets (from timestamp 3 to 10) to the recovering in chunks with the specified CHUNK_SIZE. As it is shown in the figure, the chunks of the log may arrive in different order to the recovering, so it must order them before applying the changes contained in the writesets. Of course in the meantime, the replicas are processing client requests. In the example they have validated transactions with timestamps 11 and 12.

When the coordinator detects that the next set of writesets that must be sent is small enough, it multicasts in total order the PreOnline message to set the synchronization point. When recovering delivers the PreOnline message, it starts caching all new incoming writesets from transactions that need to be validated using the replication protocol, in this case the writeset of transaction with Timestamp = 13. When the recoverers receive the PreOnline message, they send to the recovering the LastChunk message with all the missing writesets of transactions validated to that point in time (transactions with timestamps 11 and 12). From this point on and after applying the writesets in the log and validating the cached transactions, the recovering can behave as an online replica.

4.5.3. Recovering a Replica Sending the Entire Database

This section describes how the recovery process is done when the whole database must be sent to the recovering replica. In this case, the recovery facilities provided by the DBMSs are needed to recover the state of the database. This scenario can be raised when it is necessary to introduce a new replica in the cluster to improve performance of applications or when the recovering replica was offline for a long period of time (days, weeks...) and the amount information contained in the writesets of the application server log to send is bigger than the database size.

In those scenarios, it is faster to recover the new replica not by sending the whole huge set of writesets contained in the application server log, but sending a snapshot of the current database state and only a relatively small set of writesets at the end of the process. In particular, the required writesets to send from the application server log will only be those of transactions committed between the creation of the database snapshot and the end of the recovery process.

The previous protocol can be extended easily to incorporate a solution when a huge set of writesets must be sent. The pseudocode of the protocol is also shown in Figure 4.9. The process is as follows. When the coordinator receives the StartRecovery message, it checks –by means of the *isTooOld(timestamp)* function– if it is worth to send the whole database instead of the application server log (lines 43-47). If this is the case, the coordinator creates a file containing an snapshot of the current database. This can be done using the recovery facilities –such as the point in time recovery (PITR) [Posa]– provided by the DBMSs. For example, PITR allows to get/set the state of a database as it was at a particular moment in the past.

When the database snapshot has been created, the coordinator sends a DBBackup message to the recovering attaching the snapshot (line 46). In this step, the other recoverers do nothing. When the recovering has finished to receive the whole database snapshot, it starts the new database (lines 6-11). The new database contains a new timestamp in the DBState table

identifiying the last transaction committed. The timestamp is used as a new synchronization point to trigger the process described in Section 4.5.1 sending again the StartRecovery message. Thus, from this point on, the recovering process can be finished using the same protocol described above.

4.5.4. Example

Figure 4.10 shows an example scenario in which the coordinator of the recovery process decides to send the whole database.

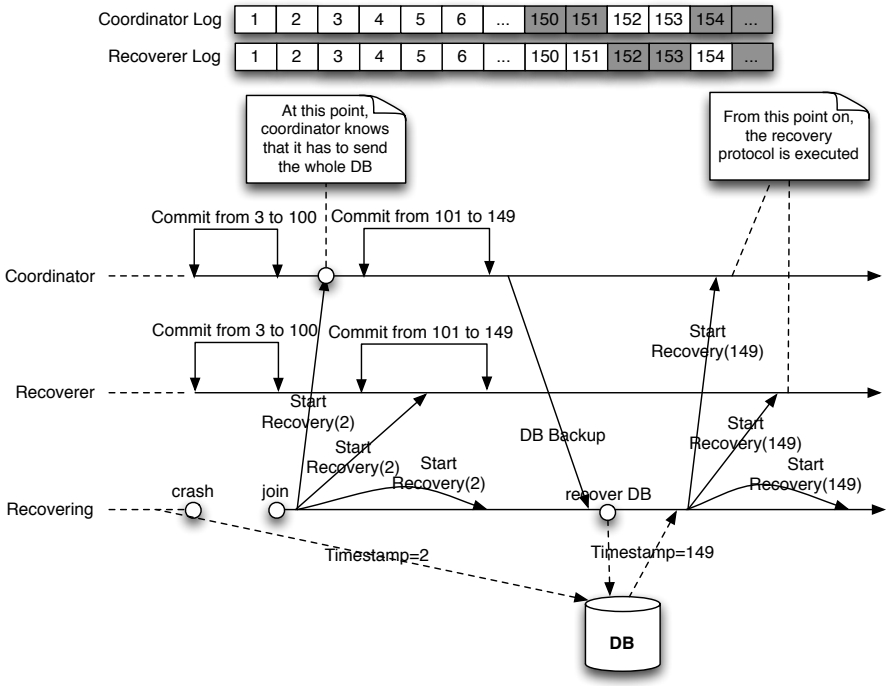


Figure 4.10: Parallel recovery sending the whole database

As in the previous example, a replica crashes after committing the transaction with Timestamp = 2. When the replica re-joins the cluster as a recovering, it gets the last committed transaction timestamp from the DBState table and sends it to all online replicas attached to the StartRecovery message (Timestamp = 2).

After having received the StartRecovery message, the coordinator decides that it is better to send the whole database state than a huge set of writesets (the ones from Timestamp = 3 to the last committed timestamp at that moment, that is, Timestamp = 100). At that point in time, the coordinator creates a file containing an snapshot of its current database state. In the meantime, each online replica continues processing client requests and therefore committing transactions. When the database snapshot is finished, the coordinator sends it to the recovering (DBBackup message). In the database snapshot is contained –in the DBState table– the timestamp of the last committed transaction at the concrete point in time when the snapshot was created at the coordinator (Timestamp = 149).

When the recovering receives the database file, it performs the recovery process at database level. When the database recovery process has finished, the recovering reads from the DBState table the last timestamp of the recently installed database (Timestamp = 149 in the example) and sends again the StartRecovery message. From this point on, the recovery protocol continues sending chunks of the application server log (Depicted in gray on the top of the figure) as is described in Section 4.5.1.

4.6. Evaluation

This section describes the evaluation performed to the replication and recovery prototypes implemented. The evaluation has been performed in a cluster of 10 machines connected through a 100 Mbps switch. The nodes have 2xAMD Athlon CPUs (2GHz), 1 GB of RAM, two 320 GB hard disks and run Fedora Linux. Each replica consists of one JOnAS v.4.7.1 application server [Objf] and a PostgreSQL v.8.2.1 DBMS [Posb]. In order to communicate the replicas, JGroups [JGr] group communication system has been used in the evaluation of the replication protocol, whilst Spread [Spr] has been used in the evaluation of the recovery protocol. In order to distribute the workload generated among the replicas, the Apache HTTP server [Apab] and its mod_jk module have been used.

The dealer application of the SPECjAppServer benchmark has been used to drive the evaluation process of the protocols. SPECjAppServer [Sta04] is a new benchmark to measure the performance of J(2)EE application server implementations, developed by the Standard Performance Evaluation Corporation (SPEC). SPECjAppServer has replaced the ECPerf benchmark [Sun03a] as the reference benchmark in the industry for J(2)EE technology evaluation. SPECjAppServer 2004 is the current version and it has been developed to use and stress the main services provided by recent J(2)EE application servers such as cache, persistence, transactions, and load balancing among others.

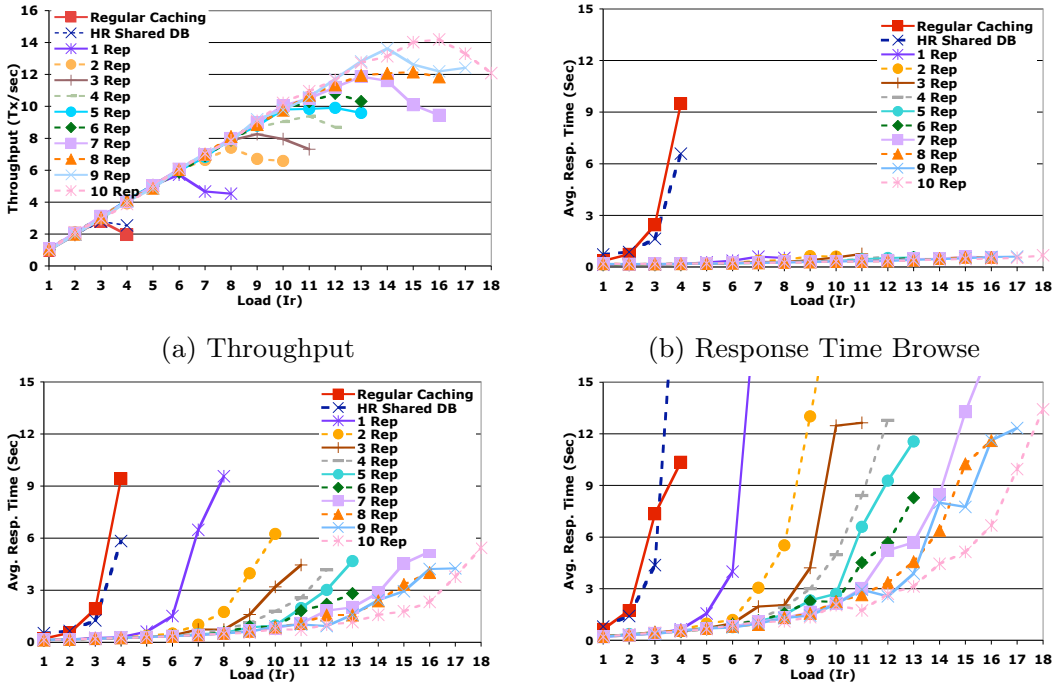
The SPECjAppServer's dealer application includes a workload generator that emulates automobile dealers interacting with the system through HTTP. Each dealer performs several synchronous requests to the application. This means that a new request is not sent until the response of the previous one has been obtained. In SPECjAppServer, there are three different types of transactional requests: purchase vehicles (25%), manage customer inventory (25%), browse vehicle catalog (50%). Browse transactions are read-only, purchase transactions have a significant amount of writes, and management transactions exhibit the highest fraction of updates. The main parameter in the tests is the injection rate (IR), which models the injected load. The number of clients is IR $\times$ 10. The SPECjAppServer target determines, for a given load, the conditions that the performance metrics of the system should fulfill. For example, it specifies a maximum response time for all requests (2 seconds). Furthermore, the response time corresponding to the 90% percentile may be at most 10% higher than the average response time. The throughput is measured as the business transactions completed per second (Tx/sec).

The next sections detail the evaluation performed on the replication and recovery protocols.

4.6.1. Replication Protocol

First of all, the results of the replicated multi-version cache have been compared with the traditional caching of JOnAS (no replication) and a replicated application server (JOnAS) with

2 replicas sharing a single database (horizontal replication) where only stateful session beans are replicated.



(a) Throughput

(b) Response Time Browse

(c) Response Time Purchase

(d) Response Time Manage

Figure 4.11: SPECjAppServer Results

4.6.1.1. SPECjAppServer Benchmark Results

Figure 4.11 (a) shows the overall throughput with increasing loads. The figure shows curves for traditional caching without replication, horizontal replication with 2 replicas (HR Shared DB) and our approach for 1-10 replicas. The first noticeable fact is that traditional caching and horizontal replication can only handle a load up to 3 IR. In contrast, our replicated multi-version cache outperforms these two implementations by a factor of 2, even if there is only one replica. The reason is that the multi-version cache is able to avoid many database reads compared to regular caching. Horizontal replication did not help because the shared database was already saturated with two application server replicas. With 3 replicas (not shown), the system deteriorated and did not even achieve an IR of 1. The replicated multi-version cache is able to handle a load up to 14 IR, achieving a throughput of 14 Tx/sec when using 10 replicas. Comparing this result to the load of 6 IR and throughput of 6 Tx/sec achieved with a single replica (and also to 3 IR - 3 Tx/sec obtained with traditional caching), it is clear the advantage of scaling out the cluster using vertical replication. That is, by adding new replicas a higher number of clients can be served.

At the beginning, adding a new replica will increase the throughput by 2 Tx/sec, after a certain number of replicas the increase is 1 Tx/sec. From nine to ten replicas the gain is around 0.5 Tx/sec. The reason is that changes performed by update transactions have to be

applied at all replicas. By increasing the load, each replica spends more time applying changes and has less capacity to execute new transactions. Nevertheless, the scale out achieved with our approach by far outperforms horizontal replication.

Even when the replicated cache configurations saturate (that is, when the throughput is lower than the injected load), the configurations with a higher number of replicas exhibit a more graceful degradation. For instance, for IR = 13, both the 5-replica an 8-replica configuration are saturated. However, the achieved throughput with 8 replicas is higher than with 5 replicas, providing clients a better service. This is very important, since it will help the system to cope with short-lived high peak loads without collapsing.

Figure 4.11 (b-d) show the response time for browse, purchase and management transactions when the load injected is increased. Interestingly, read-only transactions (browse transactions) are not affected by the saturation of update transactions. As can be seen in Figure 4.11 (b) the response time curves are almost flat independently of the number of replicas even at high loads when the system reaches saturation. The reason is that for read-only queries our application server caching is very effective avoiding expensive database access in many cases. Also, read-only transactions do not require communication among the replicas. It can be observed that both, regular caching and horizontal replication, saturate with IR = 3, since the response times increase exponentially for browse transactions.

Purchase transactions (Figure 4.11 (c)) are quite different since they are update transactions. The response time for all configurations reaches saturation (it grows exponentially) at some time point. The response times for traditional caching and horizontal replication are worse than for the multi-version approach even for low loads showing that our caching strategy saves expensive access to the database. Furthermore, the replicated architecture provides low response times until saturation is reached. Finally, the more replicas the system has, the more graceful is the degradation of the response time at the saturation point. This is important since acceptable response times can be provided in case of short-lived peaks.

This behavior is even more noticeable in the case of manage transactions, which have the highest percentage of updates (Figure 4.11 (d)).

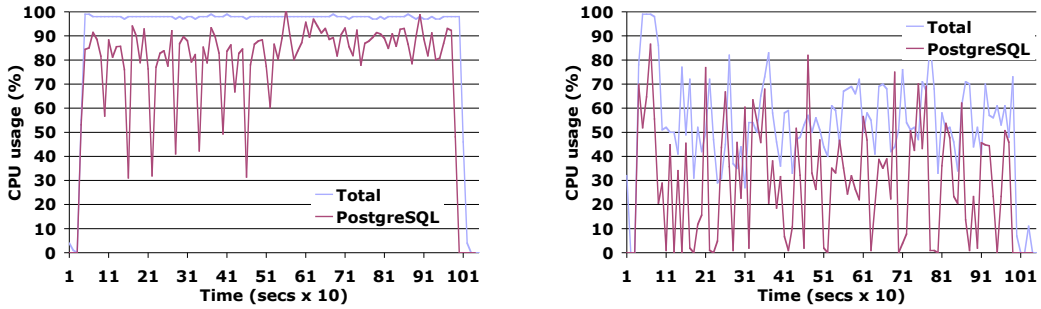
The different behavior of the purchase transactions compared to browse transactions has to do with the fact that update transactions propagate their changes to all the replicas in the system, and also have to write changes to the database. Thus, a higher overhead is created leading to worse response time. This behavior is even more noticeable in the case of manage transactions, which have the highest percentage of updates (Figure 4.11(d)). Again, however, degradation of response times is more graceful with larger number of replicas.

4.6.1.2. CPU Analysis

This section examines the CPU usage of the database and the application server during 16 minutes of executing the benchmark. Each of the following figures shows two curves. One curve is the CPU usage of the database and the other is the overall CPU usage. The gap between the two curves is mostly the application server (and replication protocol) CPU usage.

The results for regular caching and our multi-version cache with a single replica for IR = 4 are shown in Figure 4.12. At this load, the system is saturated with a 100% usage of the CPU with 1 replica and regular caching (Figure 4.12(a)). The database consumes most of the CPU processing time. There are depressions in the utilization graph of the database. They have to do with the way PostgreSQL handles updates. Periodically, when buffers are full,

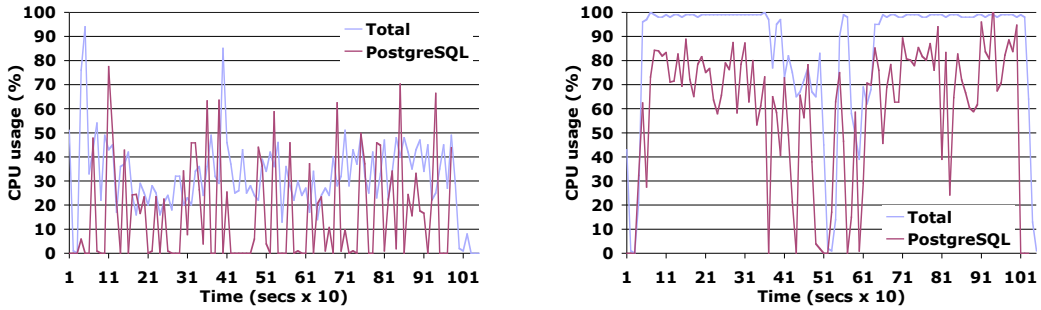
it stops transaction processing and forces data to disk. This results in underusing the CPU. The single replica multi-version cache configuration shows a significantly smaller CPU usage (Figure 4.12(b)). The CPU usage of the database is much smaller due to the multi-version cache. This saves database access and reduces the CPU resources required by the database instance. Thus, the system is not saturated for IR = 4.



(a) Regular caching

(b) Multi-version cache

Figure 4.12: CPU usage: One replica, IR = 4



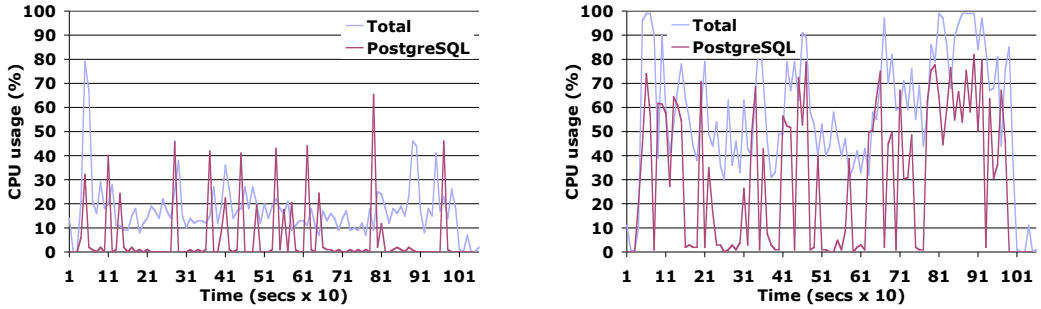
(a) IR = 4

(b) IR = 10

Figure 4.13: Multi-version cache. CPU usage: Two replicas

Examining the 2-replica configuration of our replicated cache for IR = 4 (Figure 4.13(a)), the results are quite different. Although there are some high peaks in the CPU usage, the area covered is much smaller than for the 1-replica configuration. The overall CPU usage has been significantly reduced. This means that for the same load the overhead at each replica is smaller, resulting in an effective sharing of the load. In the 6-replica configuration and IR = 4 (Figure 4.14(a)), CPU usage is even further reduced with a very low amount of CPU devoted to the database. This explains the scalability of our approach. The more replicas in the system, the better the load is distributed.

Figure 4.13(b) shows the 2-replica configuration when it is saturated at IR = 10. At this load, the database usage of the CPU amounts to 80% which means that the database is the bottleneck. The 6-replica configuration is not saturated in this setting (Figure 4.14(b)) since the CPU usage of the database is lower. This confirms the effective distribution of the entire load (application server and database load) among replicas, which results in the scalability of



(a) IR = 4

(b) IR = 10

Figure 4.14: Multi-version cache. CPU Usage: Six replicas

the approach.

Another important conclusion is the efficacy of collocating application and database server on the same node which distinguishes our vertical replication approach from previous solutions. It enables adapting the CPU resources needed by each kind of server without replica configurations. The operating system takes care of distributing CPU to the servers according to their needs.

4.6.1.3. Profiling Tool Results

A profiling tool called JProfiler [EJ] has also been used to analyze the differences in response time between the application server with and without the multi-version cache. It measures both the replication overhead and the savings obtained by the multi-version cache. Since the profiling tool introduces a very high overhead, the profiling could only be done with a single replica and the lowest IR of 1. The results show the overall number of seconds spent during the whole experiment on methods with different functionalities (Table 4.1). The group communication system (JGroups) and the replication classes introduce a non-negligible overhead as expected. However, it must be noticed that read only transactions (50% of the load) are not affected by this overhead. The multi-version cache compensates the replication overhead by improving the caching efficiency and reducing the database access (rows at the bottom) in a 27.6% and 51.5%, respectively.

	No Replication	**Replication**	**When**
JGroups		130,00	update tx
Replication Classes		143,00	update tx
Entity Serialization		3,20	update tx
SFSB Serialization		4,91	SFSBs update
Entity bean caching	152,00	110,00	always
DB Access	227,00	110,00	always

Table 4.1: JProfiler Results

4.6.1.4. Scalability Analysis

In this section has been measured the scalability of the replicated cache, that is, how much the load can be increased when increasing the number of replicas. To measure the scalability, the response time (RT) threshold of the SPECjAppServer benchmark has been taken –2 seconds– and then it has been observed for each configuration (e.g. number of replicas) the maximum load (IR) for which the response time remained below the 2-second threshold. Additionally, in order to observe the behavior under peak loads, the maximum load for a 5-second threshold has been also measured.

Figure 4.15 shows the scalability results. For browse transactions there are not shown any curves since for all tested replica configurations and injected IR the response time was well below 2 seconds. For purchase transactions, it can be seen that for small configurations the sustainable load increases sharply when increasing the number of replicas, while it only increases slightly when there are already many replicas in the system. This means, 5 replicas are able to manage a total of 110 clients (IR = 11), that is, an average of $110/5 = 22$ clients per replica. 10 replicas manage 150 clients (IR = 15), that is, an average of 15 clients per replica. Still, scalability is considerably good considering the substantial fraction of updates involved in purchase transactions.

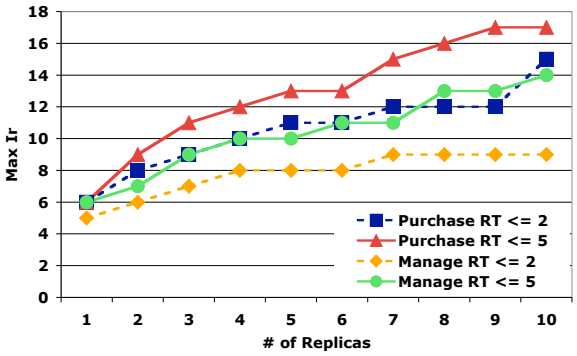


Figure 4.15: Scalability Analisys

For manage transactions the system does not scale as well as for purchase transactions. This is expected since manage transactions have a higher percentage of updates resulting in a higher replication overhead. For this kind of transactions the system does not scale beyond 6 replicas for the 2-second threshold. However, if the tolerance to peak loads (threshold $RT <= 5sec.$) is observed, having additional replicas is beneficial. Manage transactions with a threshold $RT <= 5$ scale almost as well as purchase transactions. That is having 10 replicas, the system can still provide reasonable response time (below 5 seconds) at high loads, while this is not the case for 6 replicas.

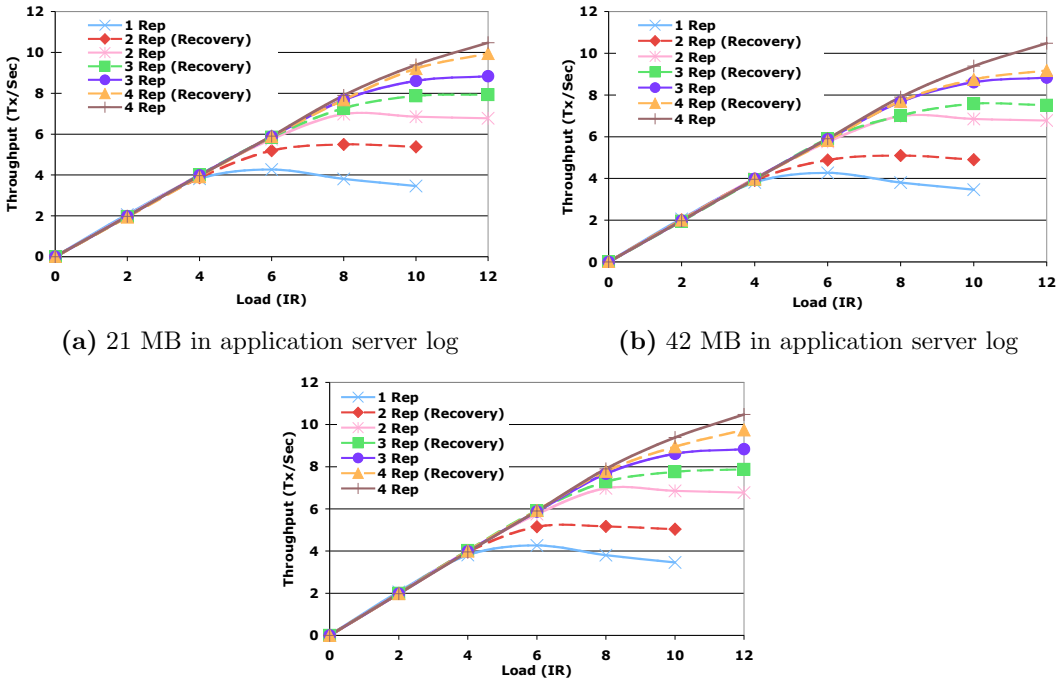
4.6.2. Online Recovery Protocol

This section describes the evaluation of the online replication protocol. The SPECjAppServer 2004 benchmark [Sta04] has also been used as test application for the recovery protocol. Different injection rates (IRs) have been used to observe the behaviour of the cluster

under different loads during recovery. The size of the database used by the SPECjAppServer has been fixed to 50 MB. With regard to the size of application server log messages sent during the recovery process, we have experimentally tested that 32 KB-messages is the best size for Spread GCS in terms of delivery time. As each writeset has an average size of 4.38 KB, every Spread message contains about 7 writesets of the application server log. In all the experiments, the recovering replica is incorporated to the cluster after a certain number of transactions (10,000) have been committed to the database to warm up the experiments. The execution of these transactions implies that additional writesets are added to the application server log before the start of the recovery process. Finally, when the recovery process finishes, the cluster is reconfigured to incorporate the new replica in order to process new client requests.

4.6.2.1. Throughput

The goal of this experiment is to measure the impact of the recovery in the throughput. For this purpose, we have run the SPECjAppServer with an increasing load (IR) and a different number of replicas (1-4). These sets of experiments provide the maximum throughput of the replicated system. The results are shown in Figure 4.16.



(a) 21 MB in application server log

(b) 42 MB in application server log

(c) Sending the database (50 MB)

Figure 4.16: SPEC throughput with and without recovery

A baseline for the experiment has been obtained for comparison purposes. The baseline has been obtained using clusters with 1, 2, 3 and 4 replicas without triggering the recovery protocol. The curves of the baseline are shown in the tree figures. In order to study the impact of the recovery process, the recovery protocol has been tested in clusters of 2 nodes (with one recovering replica and one recoverer replica), 3 nodes (one recovering/two recoverers)

and 4 nodes (one recovering/three recoverers) with different configurations. The expected throughput of a recovery configuration should be in between the configuration with one replica less and the one with all nodes running. That is, the throughput of a configuration with 2 recoverers and one recovering should be higher than the one with 2 replicas and smaller than the one with 3 replicas.

Figures 4.16 (a) and (b) show the throughput of clusters that perform the recovery process using only the application server log. The difference between both figures is the amount of writesets stored in the application server log: 5,000 writesets (about 21 MB) in (a) and 10,000 writesets (about 42 MB) in (b). When comparing the results in figures 4.16 (a) and (b), it can be observed that the throughput of the recovered clusters –independently of the number of replicas in the cluster– is higher when the amount of application server log sent is lower. Of course, as it will be seen later, this is because the duration of the recovery process is lower when less writesets are sent.

For the recovery curves in each figure, there is a point in which the throughput starts to become flat when increasing the IR. This means that the cluster has reached the saturation point and can not process more client request whilst performing the recovery process. It is also observed that the more replicas are in the cluster the greater is the throughput reached. In the experiments performing the recovery process, the saturation points are reached at IR = 6 for two replicas (1 recoverer/1 recovering), at IR = 10 for 3 replicas (1 recoverer/2 recovering) and at IR = 12 for 4 replicas (1 recoverer/3 recovering).

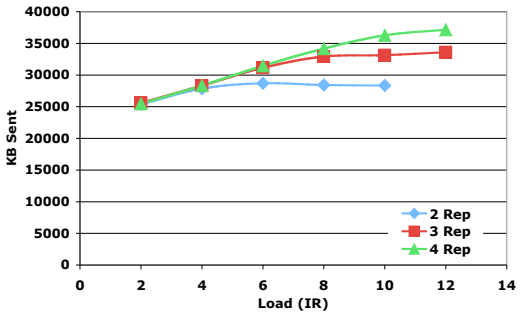
It can also be seen that the throughput for a concrete cluster configuration where the recovery process has been triggered (e.g. 2 Rep (Recovery) curve) is always below the one with the same number of replicas but without performing recovery (2 Rep curve) –this is due to the overhead introduced by the the recovery process– and above the one with one replica less (1 Rep curve). This is because at the end of the recovery process, the cluster has been scaled with the replica recently recovered. With the new replica, the load injected in the cluster can be better balanced, obtaining thus better response times for the client requests and therefore, a better overall throughput.

In Figure 4.16 (c), the recovery process is done by sending the whole database state (about 50 MB) plus the required part of the application server log that is necessary to complete the synchronization of the application server state. In total, the amount of information required in the experiments of this scenario is similar to the amount of information required in the scenario where 42 MB are stored in the application server log (Figure 4.16 (b)). Thus, comparing figures 4.16 (b) and (c) –also independently of the number of replicas in the cluster– it seems that, for this log size, it is worth to perform the recovery process sending a database snapshot instead of sending a large amount of writesets from the application server log.

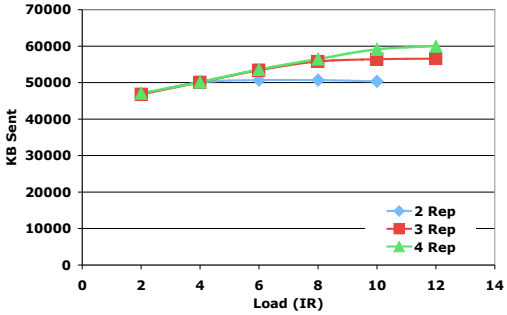
4.6.2.2. Data Sent

The goal of this experiment is to get a deeper view of the results of the previous experiment. For this purpose, we measure the amount of data that is sent as part of the recovery process. Figure 4.17 shows the curves corresponding to the number of KB sent from the recoverer replicas to the recovering replica in order to bring it online in the different scenarios.

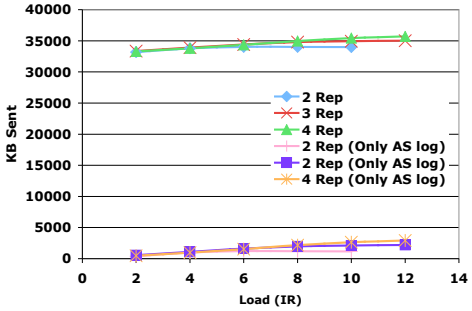
Whilst the recovery process is being performed, online replicas are also processing client requests. In each graph of Figure 4.17, when the IR is increased, there is a point where the different curves become flat, what means that the same amount of information has been sent



(a) 21 MB in application server log



(b) 42 MB in application server log



(c) Sending the database (50 MB)

Figure 4.17: KBs sent to the recovering replica

during the recovery despite that the load injected was greater. This means that the system has reached the saturation point and has not processed more client requests. Therefore, the amount of data sent is the same. The saturation points in this figure match with the points in Figure 4.16 where the throughput also starts to stabilize.

When comparing figures 4.17 (a) and (b) it is clear that the amount of information sent to the recovering replica is almost the double when the log size changes from 21 MB to 42 MB. This justifies that the throughput for each cluster configuration in Figure 4.16 (b) is lower than in Figure 4.16 (a).

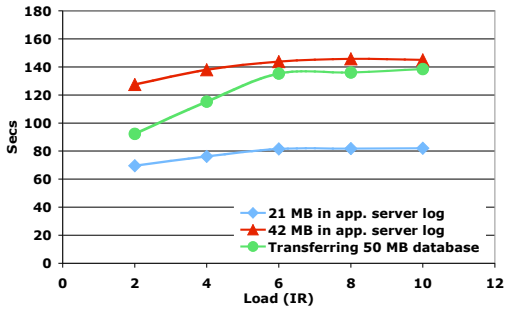
Figure 4.17 (c) shows the results obtained when sending the database snapshot plus the required part of the application server log. Before sending the database, its information is compressed (from 50 MB to 31 MB). There are shown the curves for experiments with 2, 3 and 4 replicas including the recovering replica. The curves that are marked as "Only AS log" show for the same configurations, only the amount of data (KBs) required from the application server log (that is, without taking into account the 31 MB of the compressed database).

If figures 4.17 (b) and (c) are compared, it can be seen that in order to recover a replica for this log size, the amount of information to transfer is lower sending the whole database plus a small part of the application server log than sending all the information in the application server log. This demonstrates the improvements in the throughput in Figure 4.16 (c) with regard to Figure 4.16 (b). Thus, sending the whole database is worth it when the amount of information contained in the application server log is high, about the same size as the size of the database in use. Moreover, if the throughput in figures 4.16 (a) and (c) are compared, it is observed that the results are similar. This can be explained because, as it is shown in figures

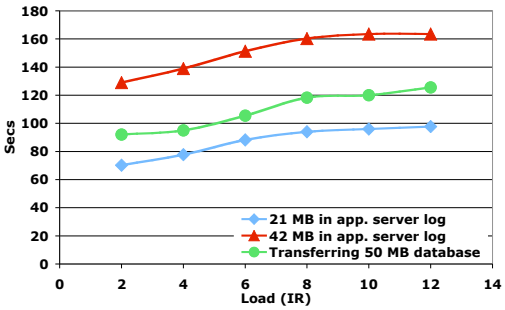
4.17 (a) and (c), the information sent to the recovering replica is also similar thanks to the compression process of the database.

4.6.2.3. Recovery Time Analysis

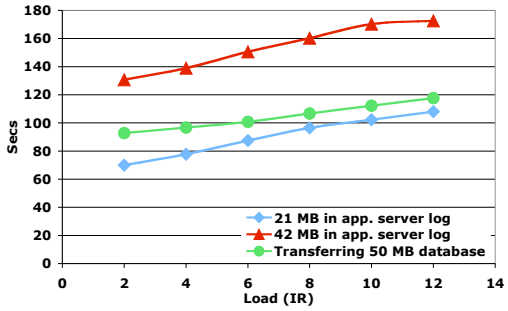
The goal of this experiment is to measure the duration of the recovery process. Figure 4.21 shows the duration of the recovery process from the point of view of the recovering replica in the three different configurations used, that is, sending to the recovering replica application server logs of 21 MB and 42 MB, and sending the database. Each sub-figure shows the results obtained varying the number of recoverer replicas from 1 to 3.



(a) 1 Recovering/1 Recoverer



(b) 1 Recovering/2 Recoverers



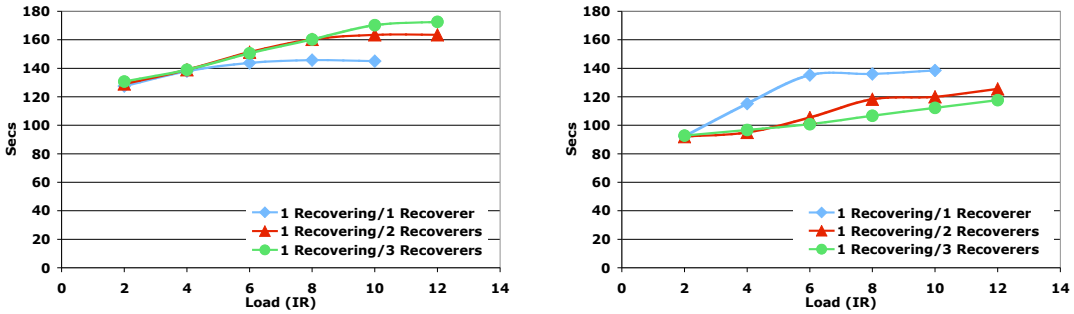
(c) 1 Recovering/3 Recoverers

Figure 4.18: Time spent in recovery

As expected, when the IR increases, the duration of the recovery process also increases. This is due to the fact that the number of requests that process the recoverer process increases, and therefore, more log entries are sent to the recoverer.

Let's see in isolation two of the three configurations shown in Figure 4.21. Figures 4.19 (a) and (b) show respectively the recovery time curves for the experiment sending the application server log with 42 MB and the 50 MB database, varying the number of recoverers. In Figure 4.19 (a) it is observed that when the cluster is not saturated, the times obtained are more or less the same. When the cluster with 1 recoverer replica is saturated at IR = 6 (See Figure 4.16 (b)), the time is stabilized. Despite the IR is increased, from the saturation point the cluster does not process more request because the SPECjAppServer clients are synchronous. This fact can be observed in Figure 4.17 (b), where the curve of the KB sent in the cluster with 2 replicas (1 recovering/1 recoverer) from IR = 6 becomes flat. The same behaviour is

found in the cluster with 2 recoverer replicas at IR = 10.



(a) Recovery time for a 42 MB application server log

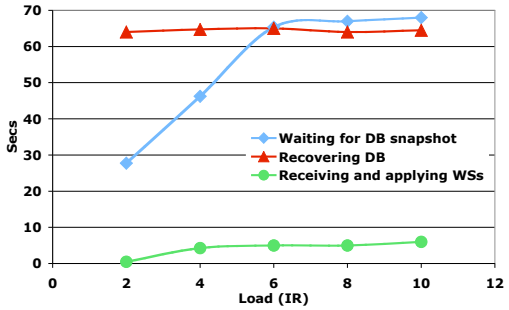
(b) Recovery time for a 50 MB database

Figure 4.19: Time spent in recovery grouped by the number of recoverers

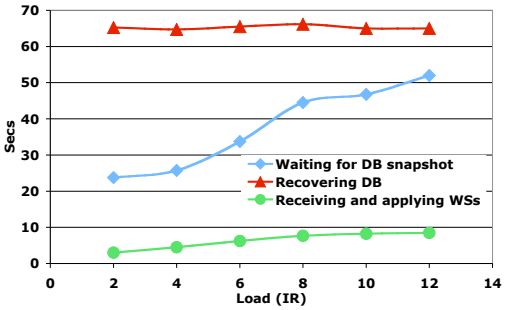
Also in Figure 4.19 (a) it is observed that when the number of recoverers increases, the time of recovery increases. As is shown in Figure 4.17 (b), this can be explained because the amount of application server log information that is sent to the recovering replica increases when there are more replicas in the cluster. The same behaviour is found in the scenario where an smaller application server log (21 MB) is sent. However, Figure 4.19 (b) shows that the opposite behaviour is found when the 50 MB database is sent to the recovering replica. That is, the time for the recovery process diminishes in the experiments including more recoverer replicas. This can be explained with the help of the following figures.

In Figure 4.20, the time spent in the recovery process when the whole database is sent has been sub-divided into three curves. The figure shows these times from the point of view of the recovering replica and for the experiments using 1, 2 and 3 recoverers. The curves show the time that the recovering replica is waiting for the database since the recovery process starts (*Waiting for DB snapshot*), the time spent in performing the recovery of the database state (*Recovering DB*) and the time to synchronize the application server state with the application server log information sent by the recoverer replicas (*Receiving and applying WSs*).

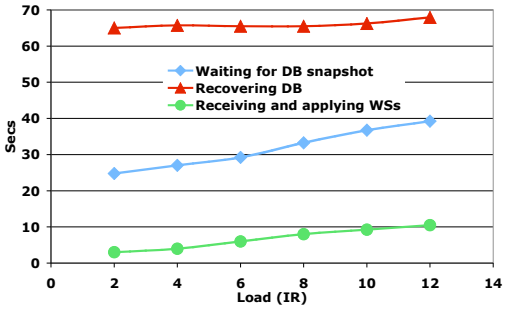
In these figures it can be observed that for each specific scenario, the time for the *Waiting for the DB snapshot* curve increases when the IR also increases until the saturation points. When there are more recoverers in the cluster, the slope of the curve diminishes. This can be explained because, for a concrete IR, if there are more replicas in the cluster, the workload that arrives to the coordinator decreases and the coordinator has more processing time to compress and send the database to the recovering replica. For example, at IR = 6 the *Waiting for DB snapshot* time diminishes about 30 seconds from the scenario with one recoverer to the scenario of two recoverers, and about 5 seconds from the scenario with two recoverers to the scenario of three recoverers. These times correspond to the decrease in time that is observed in Figure 4.17 (b) when more recoverers are in the cluster. The curve *Recovering DB* is the same in all sub-figures because the recovering replica does not do anything else and the size of the database is the same in all the experiments (50 MB). Finally, the slope of the curve *Receiving and applying WSs* increases a little bit because when there are more recoverer replicas, the cluster can process more client requests. Thus, the amount of writesets that the recoverer replicas send to the recovering replica to synchronize the state at the application server also increases a little bit.



(a) 1 Recovering/1 Recoverer



(b) 1 Recovering/2 Recoverers



(c) 1 Recovering/3 Recoverers

Figure 4.20: Time spent in recovery using the database

Finally, Figure 4.21 shows the duration of the recovery process times for different log sizes in a cluster of four replicas in which three of them act as recoverers and the fourth one as recovering. The time needed for the recovery of each log size has been measured setting an IR of 8 in SPECjAppServer benchmark. The figure shows that the growth of the recovery process time is linear with the size of the log sent to the recovering.

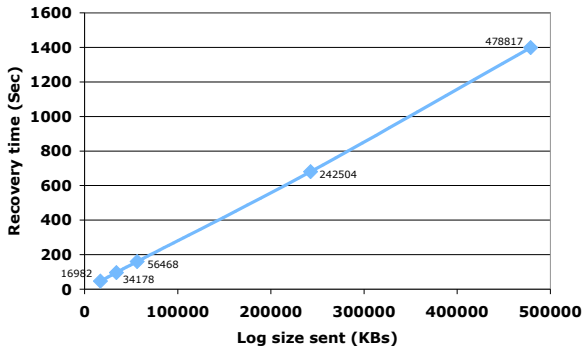


Figure 4.21: Recovery process time for different sizes of the log sent

4.7. Conclusions

This chapter has presented a replication protocol with a multi-version cache that achieves integral replication of multi-tier systems. The replication protocol takes into account both the application server and the database encapsulating the replication logic within the application server. This enables the use of off-the-shelf databases. The replicated multi-version cache provides one-copy snapshot isolation guarantees, allows the system to scale even for update workloads and takes advantage of modern snapshot-isolation databases such as Oracle and PostgreSQL. The implementation is based on an open source J(2)EE application server, JOnAS.

A thorough evaluation has been performed using the SPECjAppServer industrial benchmark for J(2)EE application server, and the results have demonstrated the good scalability of the approach.

Additionally, an online recovery protocol has been presented. The recovery protocol is complementary to the replication protocol. The protocol is able to recover the state of replicas that failed or were stopped for maintenance purposes or to incorporate new replicas to scale out the cluster. During the recovery phase, all the online replicas of the cluster continue serving client requests. At the end of the recovery, the final state of the replica that is being incorporated to the cluster is consistent with the other replicas at both tiers, application server and database. Moreover, all the online replicas in the cluster participate in the recovery process.

The results obtained from the evaluation of the prototype in the different scenarios have shown that the overhead of the online recovery process is quite affordable. Moreover, it is worth sending the whole database state to the recovering replica when the amount of information to send contained in the application server log is high enough.

CHAPTER 5

High Availability in Service-Oriented Architectures

One can be the master of what one does, but never of what one feels.

– Gustave Flaubert

Due to the rapid acceptance of web services and the faster spreading of service-oriented architectures in the enterprises, a number of mission-critical systems will be deployed as web services in next years. The availability of those systems must be guaranteed in case of failures and network disconnections. Examples of web services for which availability will be a crucial issue are those related to the security or coordination in a service-oriented infrastructure. These services should remain available despite node and connectivity failures to enable business interactions on a 24/7 basis.

In this chapter, a framework to facilitate the replication of web services is presented. The framework, called WS-Replication, relies exclusively on web service technology to allow interaction across organizations.

5.1. Introduction

Web service technology and service-oriented architectures (SOAs) are witnessing a rapid acceptance in the enterprises. If this trend continues, the new service-oriented applications will rely in certain critical services of the SOA infrastructures to accomplish their business logic. Examples of these mission critical services are those related with security, transactions, management or reliability.

Among these critical services, the transactional support for web services has been widely studied [Lit04]. Two recent efforts in this direction are Web Service Composite Application Framework (WS-CAF) [OAS05a] and WS-Coordination/WS-Transaction [OAS05b] promoted by IBM, Microsoft and BEA. These specifications provide support for coordinating transactions among different partners in SOAs using context propagation. Transactional coordination requires, among other things, the appointment of a node as transaction coordinator. The designation of a coordination entity presents an inherent weakness; if the coordinator becomes unavailable, the coordination protocol blocks and all the partners participating in the transaction can not progress in their tasks. Thus, this kind of coordination services need higher provision for availability.

However, the current SOA infrastructures still have serious lacks for providing the availability levels required by these critical services. Some of the common techniques for attaining availability consist in the use of a clustering approach. However, traditional clustering solutions are not enough in an Internet setting in which network partitions are more likely than in LANs (e.g. due to link overloads, node and connectivity) and that can separate the domain where the coordinator entity lies from the rest of the network.

In this chapter is addressed precisely the issue of web service availability in SOAs. The main contributions presented in this chapter are:

- **WS-Replication, a framework for replication of web services** - WS-Replication enables the implementation of different replication solutions based on active, passive and semi-active replication. One of the main challenges of this work is to replicate web services preserving the underlying web service autonomy principle, and avoiding the use of ad hoc mechanisms, such as the opening of non-standard ports in the nodes. Thus, the resulting systems implemented with the framework allow to provide high availability seamlessly to critical web services.

- **WS-Multicast, a multicast web service component** - The WS-Replication framework uses multicast to communicate the replicas of a web service. WS-Multicast, extends the facilities of traditional multicasts components to use SOAP as transport protocol. The WS-Multicast component can be used independently outside the framework.

To the best of our knowledge, this framework is the first one to provide high availability to any web service. Up until now, solutions were centered in replicating specific web services using ad-hoc mechanisms [SLK04, OASa]. Moreover, the WS-Multicast web service allows to communicate different nodes using the SOAP protocol.

The WS-Replication framework has been evaluated using several micro-benchmarks and a real example that uses active replication to provide redundancy to a critical service, in this case WS-CAF. The replicated infrastructure of WS-CAF's services has been stressed through an example application proposed by the WS-I organization [Web] extended with the long running transaction (LRA) model described in the WS-CAF.

The remainder of the chapter is organized as follows; Firstly, Section 5.2 refreshes the required concepts to understand the chapter. Then, Section 5.3 shows the system model used for the WS-Replication framework. Section 5.4 describes the components of the WS-Replication framework. A case of study (WS-CAF replication) is presented in Section 5.4.3. The evaluation of the framework is shown in Section 5.5. Finally, conclusions are presented in Section 5.6.

5.2. Background

The framework presented in this chapter allows to provide high availability to the web services in SOAs. Web services is one of the latest technologies developed to facilitate enterprise application integration (EAI). In contrast to other middleware technologies such as J(2)EE application servers, .NET or CORBA, web services provide a higher level of abstraction for application development, being independent from any particular development platform.

W3C offers a good definition of what a web service is[1]: *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."*.

There are three basic specifications for web services: WSDL, SOAP and UDDI. The Web Services Description Language (WSDL) [W3C01] allows to define web service interfaces. The purpose of WSDL is similar to interface definition languages (IDLs) of middleware platforms such as CORBA or COM/DCOM component models. However WSDL is not dependent on any particular platform. WSDL uses XML to define its structure. SOAP [W3Ca] is a protocol for exchanging XML messages through networks. This makes SOAP messages human readable compared to the binary format used by CORBA, DCOM or RMI messages. SOAP can use almost any underlying protocol as transport (e.g. HTTP, HTTPS, SMTP or JMS). Finally, Universal Description Discovery and Integration (UDDI) [OASb] allows to define repositories that provide functionality to publish web service descriptions and to browse its contents.

Figure 5.1 shows how these technologies are used in a simple scenario. Service providers register WSDL web service descriptions in a UDDI registry (Step 1). When service consumers require a service, it queries the UDDI registry to obtain a reference of a service provider (Steps 2 and 3). Finally, the service consumer contacts the service provider and performs the required invocations on the web service (Steps 4 and 5). All these interactions occur using the SOAP protocol.

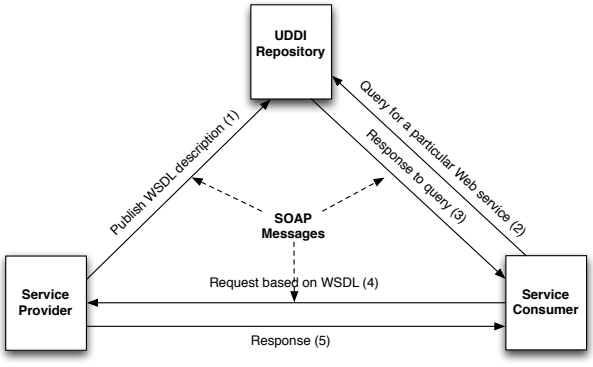


Figure 5.1: Basic infrastructure for web services

Surrounding these basic web services specifications, there are other specifications that are

[1] http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis

being standardized (the so-called WS-*, such as WS-Reliable Messaging, WS-Security, WS-CAF, WS-TX...). These specifications try to re-define the additional requirements present in distributed computing –e.g. transactions, security or reliable messaging– to fit in SOAs. In order to test the replication framework, the WS-CAF's web services providing long running activities (LRA) have been replicated. WS-CAF is a framework for coordinating web services at different levels of sophistication, from context sharing to advanced transactional models.

The LRA model is designed for business interactions that can last for long periods. An LRA is structured as a set of coordinated ACID activities. When an ACID activity completes its effects are visible. LRAs provide atomicity despite relaxing isolation by means of compensation. A compensating action logically reverses the effects of an LRA (e.g. a cancellation reverses a booking operation previously completed). LRA activities can be nested. The work performed by LRAs is required to remain compensable until the enclosing LRA informs that compensation is no longer needed. The coordinator registers the compensator through the LRA protocol before the LRA activity terminates. The LRA protocol consists of three messages: *Complete*, *Forget* and *Compensate*. Complete is sent as a notification of successful completion. Compensate compensates work that has already been completed by an LRA. The information needed to compensate is kept till a Forget message is received at that time it can be garbage collected.

Replication is the main technique to provide high availability. In the WS-Replication framework, the high availability is achieved by deploying the same service in a set of nodes (replicas), so if one node fails, the others can continue providing the service. Chapter 2 describes different approaches to replication: passive, active and semi-active replication. WS-Replication allows to implement solutions based on these approaches.

In passive replication [BMST92], one of the nodes is appointed as the primary replica and the others as backups. Clients submit requests to the primary that processes them and sends the resulting server state to the backups. That is, only the primary executes client requests. Upon failure of the primary, one of the backups takes over as new primary. In this case, failover is not totally transparent from the point of view of the client. That is, the last request issued by a client may need to be re-submitted (the primary failed before replying) and a mechanism to filter duplicate requests is needed in the backups (the new primary may have already received the state produced by that request in the failed primary).

Active replication is a complementary approach. The use of the WS-Replication framework is shown through a replicated web service following this approach. In active replication [Sch90], all the client requests to a replicated service are processed by all the replicas. The requests to a replicated service must be processed in the same order in all the replicas to guarantee that they have the same state. Moreover, the operations performed by the replicas must be deterministic. That is, with the same sequence of requests, the replicas behave deterministically producing the same output. With this approach, if one replica fails, the rest of the replicas can continue providing service in a transparent way for the client, without loosing any state. Depending on the level of reliability, the client can resume its processing after obtaining the first reply from a replica, a majority of replies or all the replies. Semi-active replication, extends active replication to support non-deterministic operations.

The communication facilities of the WS-Replication framework to connect the service replicas are based on the ones provided by group communication systems (GCSs). Group communication systems (GCS) provide *multicast* communication and the notion of *view* [Bir97].

A *view* contains currently connected and active group members, that is, the replicas of the web service. Changes in the composition of a view (*member crash* or *new members*) are

eventually delivered to the framework. In the framework, the replica failures are detected by group members when a new view is delivered excluding members of the previous view.

The multicast features of the WS-Framework allow to communicate a group of replicas using the SOAP protocol. Multicast primitives can be classified attending to the order guarantees and fault-tolerance provided. *FIFO ordering* delivers all messages sent by a group member in FIFO order. *Total order* ensures that messages are delivered in the same order by all group members. With regard to reliability, *reliable multicast* ensures that all available members deliver the same messages. *Uniform reliable multicast* ensures that a message that is delivered by a member (even if it fails), it will be delivered at all available members. It is also assumed that multicast messages are delivered to the sender of the message (self delivery).

Total order multicast is used to propagate requests in the active replication approach to guarantee that all replicas deliver the client requests in the same order and therefore, they reach the same state. FIFO order is used in passive and semi-active replication to send the state or the result of the non-deterministic actions. Since this information is only sent by one replica, the application of those messages in FIFO order will guarantee that all the replicas reach the same state. Depending on the consistency level needed by the application, multicast messages are either reliable or uniform.

Other two important properties of GCSs that are also provided by the WS-Replication framework are *primary component membership* and *strong virtual synchrony* [CKV01]. In a primary component membership, views installed by all members are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one member that survives from one view to the next one. Strong virtual synchrony ensures that messages are delivered in the same view they were sent (also called *sending view delivery*) and that two members transiting to a new view have delivered the same set of messages in the previous view (*virtual synchrony*).

5.3. System Model

The replication schema developed in order to demonstrate how the WS-Replication framework works, follows an active replication approach. The replication model is depicted in Figure 5.2.

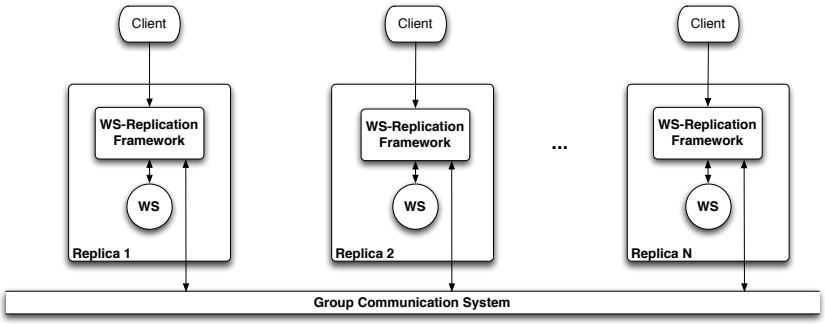


Figure 5.2: Replication model

In this approach, each replica contains the services of the WS-Replication framework and

the web service (*WS*) or the set of web services that require high availability. The set of N replicas is distributed in separate nodes across a LAN or a WAN. Each client is connected to a particular replica and sends its requests to it. Upon receiving a client request, the replica propagates it to the other replicas (including itself) through a group communication system (GCS). When the request is delivered, the request is processed in each replica. Finally, the replica that originally received the client request, returns the response to the client.

In the system model, only crash failures of the replicas are considered. A replica is considered crashed if the client can not reach its web service replica. Upon failover, the client will try to re-connect with the replication framework to get a response from another replica. Moreover, the failure of the primary replica is transparently masked to the clients. Upon failover, the processing continues at the point where the previous replica was when it failed.

5.4. WS-Replication: A Replication Framework for Web Services

WS-Replication is a framework for seamless replicating web services. That is, the framework respects web service autonomy and provides transparent replication and failover.

The communication within the framework is exclusively based on SOAP without forcing the use of an alternative communication means. WS-Replication allows the deployment of a replicated web service using web service technology and SOAP for transporting information across nodes. What is more, clients invoke a replicated web service in the same way they invoke a non-replicated one. Internally, WS-Replication takes care of deploying the web service in a set of nodes, transparently transforming a web service invocation into a multicast message to replicate the invocation at all nodes, awaiting for one, a majority or all replies, and delivering a single reply to the client. WS-Replication also deals transparently with failures of replicas.

Let us see what happens when a web service is replicated using the framework. Figure 5.3 depicts the high level vision of WS-Replication. In the figure, a replicated web service, *WS*, is deployed in two replicas that use the active replication approach provided by the WS-Replication framework. The figure shows the web service's WSDL interface (shown as a rectangle with the name of the web service, *ws*), its proxy (shown as an ellipse) and its implementation (shown as a circle with name *WS*). The client is local to the replica on the top of the figure[2].

The client invokes a web service interface (*ws*) with the same operations as the target web service (step 1 in the figure). However, the WS-Replication framework internally intercepts the request through a proxy (step 2) and multicasts in total order the requests to all the replicas (including itself) using a private transport web service (step 3). Upon receiving the multicast message at each replica (step 4), the message is processed by the framework to provide the required delivery guarantees (e.g. total order). Finally, the web service invocation is locally executed at each replica (step 5).

WS-Replication consists of two major components that are introduced in the next sections: a *web service replication component* and a *reliable multicast component* (WS-Multicast).

[2]If needed, WS-Replication also supports to deploy proxies at nodes that do not hold replicas of the web service. This is useful when clients are remote to all the replicas.

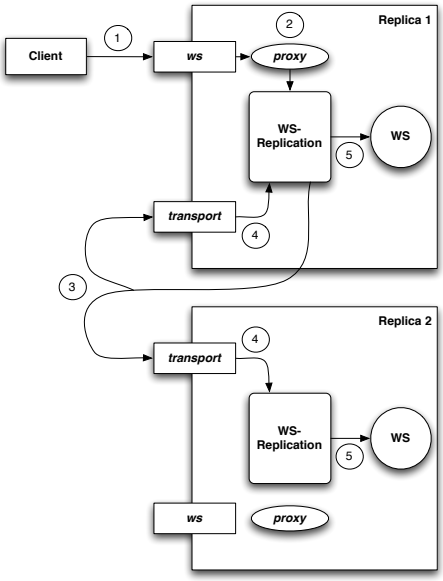


Figure 5.3: High level view of WS-Replication

5.4.1. Replication Component

The web service replication component enables the replication of stateful web services even with persistent state. In the case of active replication, the replicated web service must behave as a state machine [Sch90], that is, the service should produce the same output if the same input sequence is provided (non-deterministic operations are not supported).

The replication component consists of a *web service deployer*, a *proxy generator* for seamless web service replication, and a *web service dispatcher* (WS-Dispatcher).

The web service deployer provides a distributed deploying facility in the form of a web service that enables the deployment of a service to be replicated at different nodes from a central location. It takes a web service bundle as argument that contains all what is necessary to deploy the web service in a single file (like ear or jar bundles for Java applications) and deploys it at all replicas.

The proxy generator generates a proxy for each operation of the web service that intercepts the invocation and interacts with the dispatcher to replicate the invocation. The proxy code carries out two tasks at the node where the invocation is issued: 1) it intercepts invocations to the replicated web service and forwards them to the dispatcher; 2) upon reception of the reply from the dispatcher, it returns it to the client.

The web service dispatcher is a component that interfaces with the group communication and takes care of the matching between invocations and messages. More concretely, the dispatcher implements two functionalities. As a sender (the instance collocated with the client), it transforms web service invocations forwarded by the proxy into multicast messages using the underlying multicast infrastructure. As a receiver (collocated with a replica), it reconstructs the web service invocation from the multicast message and invokes the target web service using a particular deployment style. Depending on the required degree of dependability the dispatcher can return control to the client (via the proxy) after receiving the first reply, a majority of

replies or all replies. The treatment of replies is indicated in the deployment descriptor of the replicated web service.

There are two types of deployment for a replicated web service: as a private web service or as a Java implementation. The former enables to deploy any web service as a replicated web service, although the price of this flexibility is an extra SOAP message involved in each invocation (the one from the dispatcher to the web service). The latter only works for Java implementations of the web service, but it benefits from saving that SOAP message which is replaced by a Java method invocation, which is significantly cheaper. This (SOAP) Java invocation corresponds to step 5 in Figure 5.3.

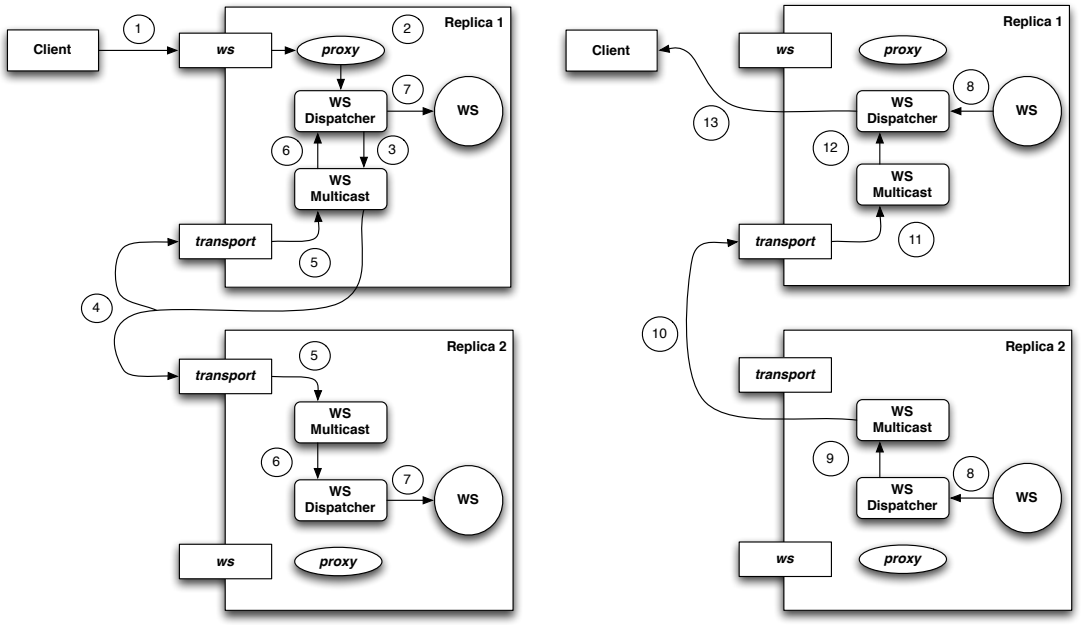
A more detailed view of the path of a replicated web service invocation is shown in Figure 5.4. The client invokes the replicated web service as a regular web service (step 1 in the figure). The invocation is intercepted by the proxy that delegates it to the dispatcher (step 2). The dispatcher multicasts the invocation through WS-Multicast (step 3). WS-Multicast uses the transport web service to multicast the message to all replicas (step 4). Upon reception of this message (step 5), WS-Multicast enforces the delivery properties of the message (ordering and reliability properties) possibly exchanging other messages between micro-protocols. When the message fulfills its delivery properties, WS-Multicast delivers it to the dispatcher (step 6). The dispatcher executes locally the target web service (step 7). The reply of the local web service invocation is returned to the dispatcher (step 8). The dispatcher where the request was originated will wait for the specified number of replies (first, majority, all). The dispatchers at other replicas use WS-Multicast (step 9) to send the reply back via unicast to the dispatcher where the request was originated (step 10). When each reply is received (step 11), the local WS-Multicast forwards the reply to the local dispatcher (step 12). The dispatcher will deliver the reply to the client via the proxy once it has compiled the specified number of replies (step 13).

Since all replicas have received the invocation and all replicas are returning the resulting reply, replica failures are masked as far as there is an available replica. That is, the replicated service will be available and consistent. The proxy will return to the client as soon as it compiles the required number of replies despite failures. Note that by collocating a proxy with the client, the client can access the replicated web service as far as there are enough replicas available.

5.4.2. Multicast Component

The WS-Multicast component provides group communication based on SOAP. More specifically, it consists of a web service interface for multicasting messages, an equivalent Java interface for multicasting and receiving messages, a reliable multicast stack for group communication and a SOAP-based messages.

WS-Multicast, as any other web service, exhibits a WSDL interface that is used to attain the reliable multicast functionality using a SOAP transport. It consists of two parts: a user interface and an internal interface. The user interface enables the creation and destruction of groups (via channels that act as group handles) and their configuration. It also allows to discover the members of a group as well as send unicast and multicast messages. The second interface is used internally by the GCS to disseminate multicast messages, perform failure detection, and other tasks to achieve group communication properties. Failure detection is achieved through a specific SOAP based ping micro-protocol that monitors the group members.



(a) Request Path **(b)** Reply Path

Figure 5.4: Low Level View of WS-Replication and WS-Multicast

A private operation of the WS is a transport operation (shown as `transport` in Figure 5.4) that takes an array of bytes as parameter, and is used to transfer messages with a SOAP transport.

The group communication stack implements the multicast functionality. In order to reuse some well-known micro-protocols such as those for total order and reliability, our SOAP group communication support has been integrated into an existing group communication stack, JGroups [JGr]. In this way, WS-Multicast can use micro-protocols of JGroups, and JGroups can be used over SOAP. A similar approach can be used with other stack-based GCSs. WS-Multicast also relies on a SOAP engine to enable web service interaction for transporting messages over SOAP. Currently there are two main SOAP engines in the open-source community: CXF [Apaa] and Axis2 [Apad]. The implementation described in this chapter is based on the early version of the Apache Axis project [Apac].

Both the public web service interface and the Java one of WS-Multicast provide the same operations. The Java interface is exactly the same as the one provided in JGroups. In order to send (receive) multicast messages to (from) a group, the sender (receiver) has to create and connect to a channel (the channel is like a socket or file handle). The parameter of the *createChannel* operation is the list of properties to configure the micro-protocols in the group communication stack. To join a group, the group member must connect to a channel providing the name of the group, after connecting to a channel. The *send* operation is used both for multicast and unicast messages. The send operation has three arguments: the message destinations, the source address and the message to be sent. If the message destination is null, the message is multicast to all members of the group. If it is not null, the message is unicast to the addresses specified. The source address indicates the recipients of the response to that

message (null indicates the channel address).

The *receive* operation is used to receive messages, views, suspicions and blocks. Internally, messages are stored in a queue. When the receive operation is invoked, the message in the head of the queue is removed and returned. This operation blocks, if there are no messages. The receive operation can return an application message, a view change, a *SuspectEvent* (indicating that a group member is suspected to be failed), or a *BlockEvent* (indicating the application to stop sending messages in order to implement strong virtual synchrony).

Finally, the *disconnect* operation removes the invoker from the group membership. The *close* operation destroys a channel instance and the associated protocol stack.

5.4.3. A Case Study: A Highly Available WS-CAF

In order to evaluate the performance of WS-Replication in a real environment, the WS-CAF's long running activity model (LRA) has been implemented[3] and replicated with the framework. The WS-CAF specification used as a reference was v.0.1 [OAS05a].

The solution implemented with WS-Replication to replicate the LRA model is based on active replication. A WS-CAF interface is provided that transparently takes care of the replication. This means that there are two interfaces of WS-CAF: a public one offered to the clients that takes cares of multicast requests to the replicated WS-CAF implementation and a private one with the real implementation invoked upon delivery of the multicast invocation. The replicated web services are those that provide the operations of WS-CTX (context management sevices), WS-CF (coordination services) and WS-TX (LRA services).

In the active replication approach invocations to WS-CAF services made by a client to the public WS-CAF interface create an invocation that it is then reliable multicast in total order using WS-Multicast to all the replicas of the WS-CAF service. The client can either wait for the first reply of WS-CAF or for a majority of responses depending on the required level of dependability. The number of replies that it has to wait for is specified in the replicated web service deployment descriptor. The public interface returns a single reply hiding the fact that the web service is replicated.

5.5. Evaluation

The evaluation has been performed in two steps. First of all, a micro-benchmark was conducted to understand the overheads of the proposed framework. In a second stage, the replication framework was used to build a replicated version of the WS-CAF's LRA model that was benchmarked using an adapted implementation of the WS-I application [Web] using long-running activities.

All nodes used in the experiments run the Linux Red Hat 9 operating system, the Axis v.1.1 SOAP engine [Apac] embedded in JBoss v.3.2.3 application server [Reda] and PostgreSQL v.7.3.2 as DBMS [Posb]. The hardware configuration at each node is shown in Table 5.1.

The micro-benchmark is based on a simple web service: a web service that implements a counter. It just takes an integer and adds it to a local counter. This simple web service is used to show the overhead of the GCS. This evaluation led to a series of improvements

[3]The LRA implementation of WS-CAF is available at ObjectWeb as part of the JASS open source project: http://forge.objectweb.org/projects/jass/

Site	# of Processors	CPU	Memory
Madrid	2	AMD Athlon 2 GHz	512 MB RAM
Bologna	4	Intel Xeon 1.8 GHz	2 GB RAM
Zurich	2	Intel Pentium IV 3 GHz	1 GB RAM
Montreal	2	Intel Pentium IV 3 GHz	1 GB RAM

Table 5.1: Hardware configuration at each node

in the WS-Replication framework (Section 5.5.1). Then, WS-Replication was evaluated and compared with two baselines: (1) the non-replicated web service with SOAP transport; (2) the replicated service with group communication based on TCP transport. Baseline (1) measures the cost of a SOAP invocation. It enables to measure the cost of replication with respect a non-replicated web service. Baseline (2) measures the cost of replicating an invocation using group communication based on TCP transport. Therefore, Baseline (2) enables to quantify the cost of introducing a SOAP transport in group communication compared to TCP-based one. This evaluation was run both in a LAN and a WAN (Section 5.5.2).

In order to show the performance in a realistic application, the aforementioned replicated version of WS-CAF was built based on the WS-Replication framework. This replicated version was used by an implementation of WS-I enriched with the LRA advanced transactional model supported by WS-CAF. The benchmark was run in a WAN (Section 5.5.3).

Four nodes located in four different countries were used for the WAN evaluations: Madrid (Spain), Montreal (Canada), Bologna (Italy), and Zurich (Switzerland). The distances among locations in terms of message latency (in ms) are summarized in Table 5.2. They have been obtained running 10 times the ping command between each location pair and taking the average among them.

	Madrid	Bologna	Zurich	Montreal
Madrid	-			
Bologna	30	-		
Zurich	41	23	-	
Montreal	130	140	123	-

Table 5.2: Average time (ms) returned by ping between pairs of endpoint locations

5.5.1. Evolution of WS-Replication

In this scenario, the counter web service is used to show the overhead of the framework. The experiment was run with a range of 1-3 replicas in a LAN. Each experiment consists of a number of 10,000 requests. There is a single client that submits a request, waits for the reply from the web service counter, and immediately after submits a new request. The results are shown in Figure 5.5.

When comparing WS-Replication (WS-Replication (v1) in the figure) with the performance of a replicated web service using TCP-based group communication (GC-TCP), it turned out that there was a huge performance degradation even for a single replica. In terms of throughput WS-Replication (v1) behaved three to four times worse than using TCP transport (GC-TCP), which it is an upper bound on the performance achievable by WS-Replication. In terms of response time, it did relatively worse, from three to five times fold increase in response

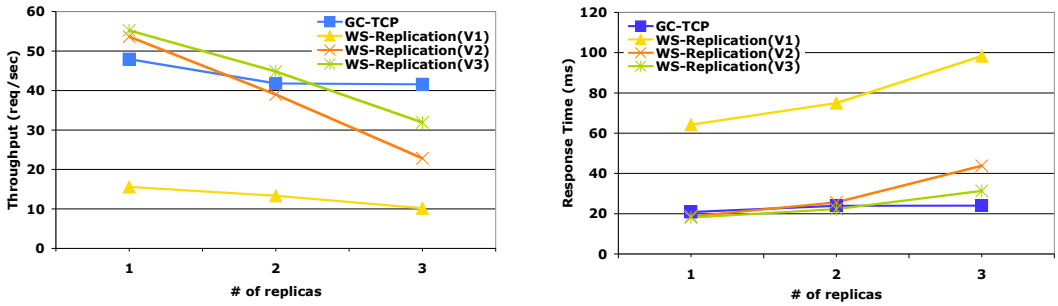


Figure 5.5: Different implementations of WS-Replication and WS-Multicast in a LAN

time with respect the TCP based group communication. The reason for the overhead was that the serialization performed at the web service level (the SOAP message) for a multicast message (send operation) was generating a 1KB message for an invocation with a single integer parameter, whilst a regular invocation with SOAP was generating a 40 bytes message. The send operation of WS-Multicast has the destination of the message, the source and the message itself as parameters. The message contains the name of the invoked operation, the parameters and the corresponding stub class. Even Java serialization was still very inefficient, although more efficient than the one of web services. A non-negligible number of bytes was being generated for a class without any attribute. The solution was to define a *streamable* interface that enabled the serialization into basic types and a multicast transport web service with a single generic parameter, an array of bytes. The alternative of rewriting the serialization methods was not enough since it did not provide enough control over the generated stream. It serialized class information that was not needed. Note that the serialization improvements are only performed on the private transport web service so they do not affect to WS interoperability. This is just an internal communication means based on SOAP transport for the replicas to communicate among them.

These improvements were evaluated in WS-Replication (v2) in Figure 5.5. It can be observed a substantial improvement, especially for one and two replicas. The throughput for two replicas was very close to the one of TCP-based group communication and increased almost four times compared to WS-Replication (v1). For three replicas, the improvement in throughput although it was significant, it showed a high degradation compared to the one showed by two replicas. The response time of WS-Replication (v2) was significantly reduced to a third of the one of WS-Replication (v1) and almost matched the one of GC-TCP, except for three replicas.

Another source of overhead was related to the fact that each web service invocation produced three web service invocations: the one made by the client and intercepted by the proxy, one to the transport operation and finally, another one to invoke the web service replica. Since the implementation of the web service is in Java, the Java deployment style of WS-Replication has been used. This deployment style enables to forward the invocation to the web service through a plain Java invocation. Of course, this performance saving only works for Java implementations. WS-Replication (v3) shows the results of this deployment style. It can be observed that the throughput increases for two and three replicas. The response time for three replicas now increases smoothly compared to WS-Replication (v2).

All these improvements (WS-Replication (v3)) prevented the bottlenecks created with 3 replicas in WS-Replication (v2) resulting in a reduction of the response time and an increased throughput.

It should be noted that with one replica WS-Replication (v2) and WS-Replication (v3) behaved better than the TCP-based group communication. The reason is that the TCP implementation creates dynamically the invocations from TCP messages, whilst WS-Replication generates specific stubs upon deployment. This resulted in better performance when there was no replication (1 replica).

It can be concluded that the throughput attained by the final version of WS-Replication was very close to its ceiling, the TCP-based group communication (GC-TCP). In terms of response time, it was almost the same as the one shown using TCP-based group communication. This means that the engineering performed in WS-Replication has been very effective in minimizing the overheads induced by group communication and SOAP.

5.5.2. Performance Evaluation of WS-Replication

The experiment in this section compares the performance of WS-Replication with the two baselines previously discussed, a non-replicated web service (SOAP in the graphs), and a replicated web service using group communication based on TCP transport (GC-TCP in the graphs). It is shown the performance from 1 to 3 replicas for an increasing load for both the TCP-based group communication and WS-Replication. The load is increased by augmenting the number of clients submitting requests. Each client submits a request and, as soon as it gets the reply, it submits a new request. The total number of requests sent by each client is $1/n$ of the overall number of requests (10,000), where n is the number of clients. The experiments have been conducted both in a LAN and a WAN setting. The client and the web service replicas were located at different nodes. In the WAN setting, the client was in Madrid (Spain) and the replicas in Bologna (Italy), Zurich (Switzerland) and Montreal (Canada). The replicated web service was configured to wait for the first response received from a replica.

In a LAN environment, it can be seen that for one replica WS-Replication performs better than GC-TCP (Figure 5.6). As explained in the previous section, this is an artifact of the experiment. The GC-TCP implementation has a generic code to manipulate and build web service invocations. This generality results in a lower performance. In contrast, WS-Replication generates specific handling code for each replicated web service that results in a lower serialization and invocation construction overheads that compensate the use of SOAP instead of TCP for one replica. That is, the overhead due to manipulation of serialization and invocations has a higher impact than the overhead introduced by the less efficient SOAP transport. The cost of using group communication and replication is shown comparing the SOAP curve with the 1-replica curves of GC-TCP and WS-Replication.

When looking at the curves for more than one replica in a LAN setting, one interesting observation is that for two replicas WS-Replication is still slightly better than GC-TCP both in terms of response time and throughput, and only for three replicas the higher overhead in the transport layer becomes the dominant factor over the serialization and invocation handling cost.

The results for the WAN environment are illustrated in Figure 5.7. In order to quantify the impact of the distance in the throughput and response time, it is shown the performance of a single replica at the three different nodes (BOLogna, ZURich and MONtreal). As one

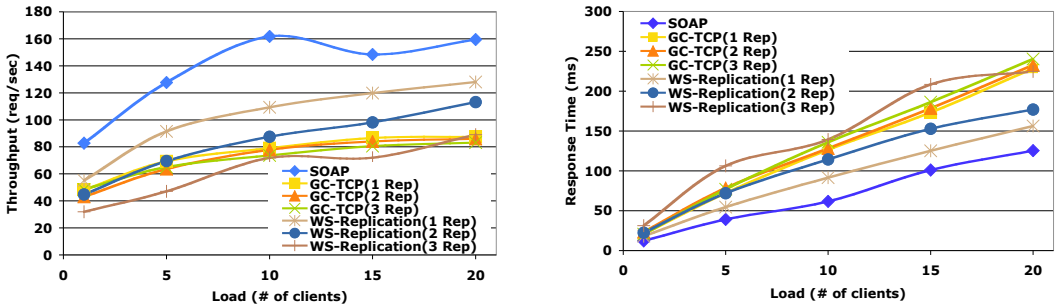


Figure 5.6: Evaluation of WS-Replication and WS-Multicast in a LAN

would expect, the farthest node from the client (located in Madrid), Montreal, yields the worst throughput and response time. The comparison of the SOAP and 1-Rep-BOL curves shows that the overhead of the replication framework with respect to SOAP is smaller than in a LAN in relative terms (from about 50% to 25%) for the closest replica.

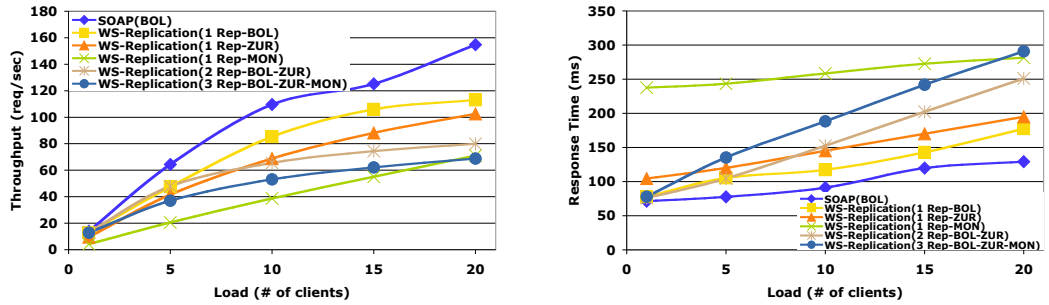


Figure 5.7: Evaluation of WS-Replication and WS-Multicast in a WAN

The overhead introduced by the coordination among replicas is shown comparing the 2-replica curve with the 1-replica ones at Bologna and Zurich. The overhead of two replicas in terms of throughput is about 20%. Looking at the overhead of a 3-replica setting, it can be seen that it has a slightly higher overhead than two replicas, but it behaves better than a single replica at Montreal. The reason is that the experiment was configured to wait only for the first reply. With three replicas, Bologna or Zurich always replied before Montreal. In terms of response time, the experiment with three replicas is significantly better than the 1-replica located at Montreal. This means that clients by using replication will get an average performance respect to the closest replicas, despite the existence of some distant replica.

As a summary, it can be concluded from the LAN experiment that the engineering performed in WS-Replication was very effective in attaining a performance competitive with a TCP-based group communication. The overhead with respect to a non-replicated web service is higher in terms of throughput, due to the extra CPU consumption spent in manipulating extra messages. However, in terms of response time is less significant. The WAN experiment shows that the relative overheads are smaller than in a LAN. What is more, it shows that a triplicated web service will behave better than a single distant replica. An important obser-

vation is that this experiment is using an almost null web service, so it measures the pure overhead introduced by extra communication. This means that using a realistic web service with some relevant processing associated to each request (e.g. performing some relevant CPU processing and possibly some IO), this performance loss will be much smaller in relative terms, as the next experiment will show.

5.5.3. Evaluation of WS-CAF Replication

In this experiment, an application based on web services defined by the WS-I [Web] is used to evaluate the performance of replicated WS-CAF's web services providing long-running activities (LRAs). The WS-I application models a supply chain management scenario. There are four roles in the application: consumers (clients), retailer, warehouses, and manufacturers. The latter three exhibit a web service interface. A retailer offers goods to consumers. The retailer places orders to warehouses to fulfill consumer requests. Orders should be served completely by a retailer. The warehouses have to keep stock levels for the items they offer. If the stock of an item falls below a certain threshold, the warehouse must refill the stock of the item from a manufacturer. Figure 5.8 shows the interactions among the different actors in WS-I.

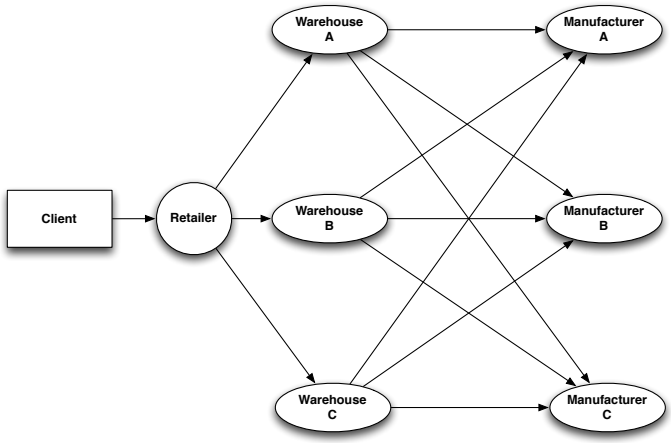


Figure 5.8: WS-I participants and interactions

The WS-I application has been enriched with advanced transactions, more concretely with LRAs, in order to evaluate our WS-Replication framework. The interaction between the client and the retailer is modeled as a top level LRA that may include several nested activities. Nested LRAs model the interaction between the retailer and a warehouse, or between a warehouse and a manufacturer. Warehouses and manufacturers are registered as participants in the active LRA when they include items in an order or when they have ordered or manufactured, respectively. These interactions can be compensated if the client interaction (top-level LRA) is not fully accomplished (e.g. if the warehouses cannot deliver the whole order to the client).

Figure 5.9 shows a potential scenario of LRA nesting in which the retailer contacts warehouse A and warehouse B. Warehouse A needs to contact manufacturer A, whilst warehouse B contacts both manufacturers B and C. The whole client request is a top-level LRA that

encompasses three nested LRAs, corresponding to each interaction between warehouses and manufacturers to refill the stock of a particular item. When a nested LRA completes, it registers the corresponding compensators with its parent. This enables compensation in case the top-level LRA does not succeed.

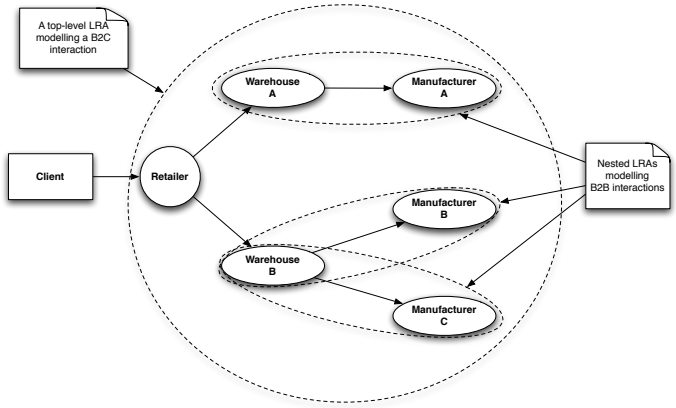


Figure 5.9: Transaction nesting in WS-I application

Figure 5.10 shows the results of the WS-I application evaluating the replicated WS-CAF implementation in a WAN setting. In all the experiments, the client and the WS-I application were run in Madrid, deployed in two different nodes. For comparison purposes, the same experiment without replicating WS-CAF was run. In this case, WS-CAF was located in Zurich (No Rep curve in Figure 5.10). Then, the experiments with one replica in Bologna (Rep-1), two replicas adding Zurich (Rep-2) and three replicas, adding Montreal (Rep-3) were executed. Each client submitted 100 requests during the experiment, plus 25 warm-up and 25 cold down requests.

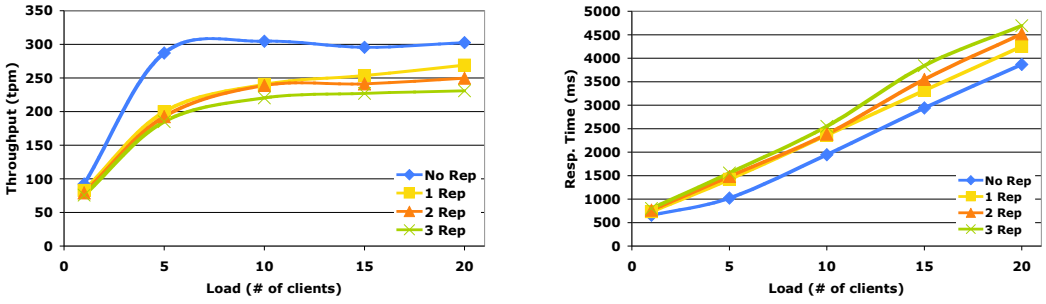


Figure 5.10: WS-I performance in a WAN waiting for the first response

It can be noted that the difference between 1 and 3 replicas is very small in terms of throughput (around 15%). This is mainly due to the stub configuration that is set up to return to the client after receiving the first reply from a replica. Awaiting the first reply is sufficient for many applications and can be considered the default case. In a later experiment, the performance results of getting the first reply and getting a majority of replies have been

compared. Comparing the throughput curves for 2-3 replicas with the non-replicated case, it can be observed that the throughput degradation is smaller for higher loads than for lower ones. The reason is that the non-replicated setting achieves its maximum throughput with 5 clients. After that, the throughput does not increase anymore. The replicated case achieves the maximum throughput with a highest load. Interestingly, the overhead of replication in terms of response time is much smaller in relative terms. The increase in response time between one and three replicas is smaller than 10% for the higher load. This is quite beneficial because the main concern in WAN replication is response time and the obtained overheads are very affordable.

In Figure 5.11 it is shown the performance difference when the replicated web service is configured to wait for a majority of the responses before returning to the client. For two replicas, waiting for a majority of replies means to wait for all the replies. As one might expect, the throughput decreases. However, the overhead is not very high especially in the case of three replicas which have 13% lower throughput. Regarding response time the relative overhead is slightly higher. This is unavoidable, since one has to wait for the second slowest reply, when waiting for a majority of responses.

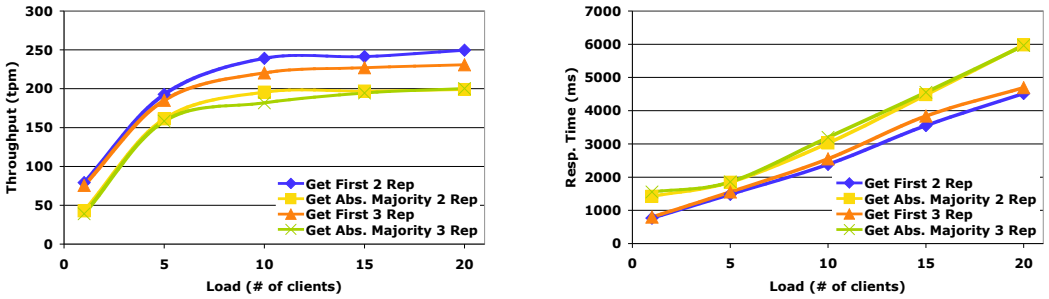


Figure 5.11: WS-I performance in a WAN comparison of first and majority responses received

5.6. Conclusions

In this chapter the WS-Replication framework has been presented. It mainly provides a set of replication facilities for seamless replication of web services. The kind of support provided by WS-Replication will be essential for the upcoming breed of critical web services with more exigent availability requirements.

The framework allows the deployment of a web service in a set of nodes to increase its availability. One of the distinguishing features of WS-Replication is that replication is done respecting web service autonomy and exclusively using SOAP to interact across nodes. One of the major components of WS-Replication is WS-Multicast that can also be used as a standalone component for reliable multicast in service-oriented architectures.

The evaluation shows that with an adequate engineering, the overhead of replication on top of SOAP can become acceptable. WS-Replication has been tested replicating a complex service, WS-CAF, and using a realistic application such as WS-I.

CHAPTER 6

Related Work

Every man takes the limits of his own field of vision for the limits of the world.

– Arthur Schopenhauer

This chapter describes the related work done in the areas related to the main topics of this Ph.D. Thesis and contrast it with the goals achieved in the previous chapters. Most of the topics are related with distributed systems, such as high availability, scalability, fault tolerance or load-balancing. A great amount of work already exists in these topics (e.g. replication techniques to provide high availability and fault tolerance have been widely studied and applied in both, academia and industry). The selected related work described in the following sections has been organized from the point of view of the two system architectures that are discussed in the thesis, that is, multi-tier and service-oriented. For each type of architecture, the scientific and industrial work in fields like data replication or caching has been examined.

6.1. Related Work on Multi-Tier Architectures

The following sections describe the efforts done in multi-tier architectures by the research community and the industry.

6.1.1. Scientific Work

The selected related work has been split on four targets: high availability, scalability, recovery and advanced transaction models.

6.1.1.1. High Availability

With regard to high availability, the Common Object Request Broker Architecture (CORBA) [Objb] encompasses many of the efforts that have been done in fault-tolerant multi-tier architectures. The Eternal was the reference system in these efforts [Nar99, MMN99b, MMN+99a, NMM02b, NMM02a]. Eternal allows to replicate application objects based on the fault tolerance properties defined by the users: replication approach (active replication or primary-backup, eager or lazy), time between checkpoints, number of replicas... Moreover, Eternal is able to detect duplicate requests when failures occur in a primary-backup approach. This mechanism used is similar to the one described for the protocols of Chapter 3. When the state of components is replicated to the backups, the primary attaches the request identifier and the response. So, after a failure in a primary replica, the new primary is able to recognize a request already processed by the previous primary and send back the response to the client. The work done in Eternal was reflected in the industry through the definition of the Fault-Tolerant CORBA (FT-CORBA) specification [Objc], being the first reference implementation. Finally, FT-CORBA was included in 2001 in the CORBA specification v.2.5. One of the major drawbacks of FT-CORBA is that non-deterministic applications are not supported. That means for example that multithreaded applications can not be fault-tolerant because they use non-deterministic system calls. This and other drawbacks of the FT-CORBA specification are described in [Nar07].

Other efforts in academia related to replication in CORBA are [FGS98, FP98, MSEL99, BM03]. However, in contrast to the replication protocols presented in Chapter 3, none of these approaches (including Eternal and FT-CORBA) addressed the integration of replication and transactions, although it was recognized as a challenging problem in [FN02].

Unification of transactions and replication in multi-tier architectures is presented in [FG01, FG02]. In these papers, Frolund and Guerraoui define formally the exactly-once correctness criterion for request processing in multi-tier architectures with stateless replicated application servers. As application servers are stateless, client requests do not modify the state in the business logic of the application, only in the database. Then, the state of the application in the middle-tier does not need to be replicated among the replicas. The authors introduce the concept of *e-Transaction* as the mean to achieve end-to-end reliability for client requests. An e-Transaction is an abstraction that guarantees two properties: (1) transaction is executed exactly once at the back-end database despite failures; (2) if a failure occurs, the clients eventually receive the results of their requests only once. To achieve this, each client request is executed as a single transaction at the server side. For each transaction a "marker" is inserted in a shared database. If a failure occurs in a replica, the new replica serving each client request during failover will look for this marker in order to ensure the exactly-once execution. The replication protocol described in their work uses a primary-backup approach at the application server (middle tier). The back-end tier is composed of multiple database instances. When a transaction is going to be committed in the database, the primary trigger a distributed commit protocol similar to 2PC. As the protocol does not assume reliable failure detection, under certain circumstances (e.g. the client wrongly suspects that the primary has failed and re-submits the request to another application server) the protocol can behave as an active replication protocol. To deal with this kind of non-determinism in the interactions, the protocol implements a consensus-like abstraction in the application server by means of a write-once/read multiple times register. The main drawback of the protocol is that it does not guarantee high availability for transactions because if an application server crashes, all the

ongoing transactions are aborted. This issue has been addressed by the replication protocols presented in Chapter 3.

In [RQC05], the authors provide another protocol guaranteeing exactly-once transactions in multi-tier architectures. The protocol is suitable only for stateless application servers and is lightweight, because it does not rely on distributed commit protocols nor exposes explicit coordination among the replicas of the application server. They achieve a lightweight protocol using only one database instance in their architecture. In order to guarantee consistency in the database, the protocol inserts a tuple identifying the client requests in a special database table before committing each transaction and relies on the database to detect duplicated requests. If the database detects an attempt to duplicate the primary key in that table, it raises an exception. The exception means that the request was already processed. When it is caught by the application server, the result of the request is retrieved from the table and sent back to the client. So they avoid the overhead introduced by the distributed commit protocol of the previous protocol. However, using this approach, the database becomes a single point of failure (because there is only one shared instance).

The same authors have recently proposed in [RRQC08] a replication protocol based on active replication tailored for multi-tier data acquisition systems. Their approach is based on that, in data acquisition systems, the replicas of the middle-tier produce output events sporadically and unpredictably in response to the receipt of a large set of input messages. Thus, in order to replicate consistently the state of the middle-tier, they do not use a coordination mechanism prior to message processing in each replica. Instead of this, they rely on a-posteriori reconciliation phase of the replicas (also without generating explicit replica coordination messages) coordinated by the back-end tier (database) only when necessary. However, their approach also relies on a centralized database. As it has been explained previously, this implies a handicap for the whole system when the database fails. Despite the authors claim that the protocol allows non-deterministic behaviour in the replicas and outperforms the state of the art of active replication schemes, it is not applicable in the same application domain than the protocols presented in this thesis.

In [ZMM02, ZMM05] is presented a transaction-aware implementation of FT-CORBA. Moreover, they study how non-determinism affects active and primary-backup replication schemes. Their protocol aims to provide end-to-end reliability (through exactly-once transactions) between the clients and the replicated servers along a multi-tier architecture. They combine horizontal replication to provide high availability at the middle tier and transactions to achieve consistency. The replication protocol uses a primary-backup approach and eliminates duplicate requests from clients with a similar technique as the one described above in the work of Frolund and Guerraoui. However, the protocol also assumes that servers have access to a logging infrastructure for storing, and retrieving, information (using a checkpointing mechanism). The log must be accessible by all the replicas of an object and support transactional operation. Thus, the centralized log and the use of a single database for the application represent two points of failure that compromise the high availability of the replicated infrastructure. As it has been commented before, these drawbacks are overcome by our protocols using vertical replication.

The integration of replication in J(2)EE application servers and a replicated database has been studied in [KJPS05]. This paper analyzes different multi-tier architectures that implement replication at application server level, at database level or both combined to provide exactly-once interactions. It exposes three different configurations explained from a primary-backup

scheme point of view (for simplicity). The first two configurations –horizontal and vertical replication– are *tightly coupled* approaches in which only the middle tier (that is, the application server) is responsible for coordinating the replication. A tightly coupled approach requires only one replication protocol that coordinates replication within and across tiers. These two configurations have been shown in Chapter 2, Section 2.4. The third configuration replicates independently the middle and the database tiers, but requires to couple both replication solutions to work properly (loosely coupled approach). This solution requires careful engineering. Then, the paper describes each relevant scenario for different configurations showing its pros and cons and failure scenarios. In this thesis we have chosen the vertical replication approach, as it is a simple and elegant solution to achieve consistency for stateful applications in the middle tier and the back-end tier at the same time using only one replication protocol.

Some other recent works have studied J(2)EE application server clustering for providing high availability to stateful applications [KMSL03, BBM+04, WKM04, WK05]. Most of this work was done in the context of the ADAPT project [ADA, BJK+05].

Kistijantoro et al. show in [KMSL03] how replication for high availability can be achieved in J(2)EE Enterprise Java Beans (EJB) containers. They explore different replication configurations similar to those analyzed in [KJPS05]: (1) horizontal replication at the back-end tier. They use proxy resource manager adaptors (JDBC drivers and XAResource interfaces) that re-issue the operations from the middle tier to all the replicas of the database; (2) horizontal replication at the middle tier. The architecture is based on an update-everywhere approach at the middle-tier with a shared database. As their approach does no use a coordination protocol for the middle-tier, in order to guarantee the consistency it uses a primary-backup approach for the resource managers. So there is only one resource manager (the primary) for the whole cluster to guarantee the serializability of transactions in the back-end tier. To ensure the consistency of the backup resource managers, the stored requests are issued to all of them. In order to maintain the consistency of the cached entity beans at each application server replica, this approach requires to re-read their state from the database each time an entity bean is accessed, what results in a loss of performance. Moreover, when a replica fails, ongoing transactions are aborted, so the solution does not provide (as our solution does) high available transactions; (3) horizontal replication at both tiers: middle tier and back-end tier. They explore this approach because the previous approaches (1 and 2) have potential single points of failure in the non-replicated tier. They explore the issues that arise when trying to implement the independent replication of both tiers using either primary-backup or active replication at the applications server tier. Finally, they claim that they can no offer a high available solution without modifying the existing services and APIs. The vertical replication approach used by our protocols eases the replication of these two tiers at the same time. Moreover, our solutions modify the application server internals but do not modify the existing services and APIs.

In [BBM+04] Babaoglu et al. describe the ADAPT replication framework. ADAPT is a generic framework that allows the implementation of different replication protocols on top of it. An implementation of the framework for the J(2)EE JBoss application server was an output result of the ADAPT project [ADA]. This framework implementation has been used to develop and test the replication protocols described in Chapter 3[1].

The authors of [WKM04] also applies the technique described in [FG02] for providing exactly-once semantics for transactions in J(2)EE application servers. However, their protocol also provides high availability and consistency to stateful applications running in J(2)EE appli-

[1] More details about the ADAPT framework can be found in Chapter 3.

cation servers. Each transaction can span a single client request, what could be a limitation for certain applications that require several client invocations inside the same transactional context. The protocol uses a primary-backup approach combined with eager replication. Thus, as we do in our protocols, when the processing of each client request is about to complete and before returning the response to the client, the changes in stateful components are replicated to the backups. However, only the state of stateful session beans (session state) is replicated. As their approach shares the database, they do not need to replicate the state of the entity beans. Only the identifiers of the entity beans are replicated to allow the recreation of the components when the primary fails. Moreover, the database is shared among the application server replicas of the cluster, so it is a potential single point of failure.

Also, the authors of the previous work describe in [WK05] the different interaction patterns that can be generated in an application server depending on the behavior of the applications. These patterns are classified depending on the number of client requests involved in a transaction and the number of transactions generated by a request (1 request/1 transaction, N request/1 transaction patterns, 1 request/N transactions pattern and N requests/M transactions). The three replication protocols presented in Chapter 3 can be matched with the patterns presented in this work. The work related to the unification of transactions and replication described above basically consider the replication of the simplest pattern (1 request/1 transaction). This pattern is also considered by our first protocol. It is widely used in web applications that require simple transactional interactions. Our second protocol deals also with the replication of multiple client request executed in the same transactional context (N requests/1 transaction pattern). This pattern is useful in applications that require conversations (e.g. to fill a shopping cart with items, a client can perform several interactions to add items to a shopping cart before to conclude the transaction). Finally, through open-nested transactions, the 1 request/N transaction and the N requests/M transaction patterns can be also used in stateful applications and thus its replication is supported by our third protocol. This pattern is used when a transaction with a long execution must be split to increase concurrency with the database. Once again, whilst our protocols still provide high availability when the back-end tier fails, for the protocols described in this paper the database is still a single point of failure for the cluster.

6.1.1.2. Scalability

In this thesis, scalability is understood as the ability of a cluster to increase its computing power adding more replicas (See Chapter 2, Section 1.6). With regard to scalability, to the best of our knowledge, we have found few studies or prototypes of replication protocols for scaling-out multi-tier architectures. The approaches described in the previous section either are suitable for stateless application servers or use a primary-backup replication scheme. This means that their scalability is limited to the capacity of the primary replica for processing client requests. The replication protocol presented in Chapter 4, that provides high availability and scalability to stateful applications deployed in multi-tier architectures, is based on the protocol described in [LKPJ05] for databases. Additionally, our protocol incorporates a middle-tier cache for persistent components (entity beans) that guarantees snapshot isolation in the local replica and, by means of the replication protocol, in the whole cluster. Thus, the following papers describe also studies related to middle-tier caches that aim to improve application performance.

The authors of [SPH06] show a "scalable" approach of a multi-tier architecture based on the JOnAS J(2)EE application server. They use an update-everywhere approach in the middle-tier, but the application servers share the database (single point of failure). The update-everywhere is achieved in the middle-tier by means of CMI stubs (Cluster Method Invocation stubs). The CMI stubs distribute the load injected by the clients among the application server replicas following a round-robin strategy. No replication protocol is used in their approach. In order to keep the consistency of stateful components in the cluster, when using CMI stubs the cache for those components provided by JOnAS must be disabled. This implies that any access to a persistent component (entity bean) must be synchronized with the database, what results in a loss of performance in the middle-tier. Thus, when comparing the number of client requests processed per second in a cluster with two replicas against the non replicated infrastructure, the improvement obtained is null. The evaluation has been performed using RUBiS [Objg], a non-standard benchmark for application servers. They achieve to slightly scale the performance of the RUBiS application in a cluster with two replicas, when they deploy the read-only components (which can have always the cache enabled) in the two replicas and partition the writable components among the two replicas. Deploying in that way the application allows also to enable the cache in the writable components. The CMI stubs perform load balancing when invocations are targeted to read-only components and redirect the request to the right application server when a potentially writable component is accessed.

In [LR03], Leff and Rayfield introduce a caching system for J(2)EE application servers. The cache is transparent for the EJB components (as in our approaches, the programmer does not writes new code to access the cache) and provides the same transactional semantics as a non-cache enabled application server. In contrast to our cache, that provides snapshot isolation, their cache provides a weaker isolation level (repeatable-read). In their architecture, application servers are replicated horizontally, sharing the database. In order to keep the caches consistent, their approach does not uses a coordination protocol among the caches. The cache consistency is guaranteed through a certification protocol that uses the shared database as a reference. At commit time, every entity bean is re-read to check whether it was modified or not. This approach has the shortcoming of all horizontal approaches since the shared database becomes a bottleneck. The certification protocol used is heavier than the validation phase of our replication protocol since it has to re-read every entity bean from the data source. The same authors propose in [LR04] to apply the EJB caching on edge-server architectures. Edge-server architectures are used in WANs to improve application performance, moving the web-content of web applications from back-end nodes to edge nodes that are placed close to the users. They test several configurations in a LAN environment simulating the latencies that occur in a WAN by means of a non-standard benchmark.

Another approach for providing consistent caching for dynamic content in web applications is studied in [AÖ06]. The architecture presents middle-tier replicas that include full-fledged web and application servers and a local cache. The coordination among the replicas is done horizontally and share a common database. Each application server sends the database queries to its local cache, rather than to the database. As in our protocol, in case of a cache hit, the query is answered without any need to contact the origin data source. In case of a cache miss, the query is sent to the database and the result is cached at the local cache. The cache stores individual tuples of the database using fine-grained locks to support concurrent accesses. A coordination protocol based on invalidation messages is used to maintain the caches consistent. When a tuple in a cache is modified, the protocol must send invalidation messages to the other

caches that hold the same tuple. Thus, each time that a cache receives a query, it must check the local copy of the table to assure that (1) the tuple is already cached and (2) the cached copy is valid. Despite this approach shares the same strong consistency goal as the multi-version cache presented in this thesis, the overhead introduced by invalidation messages does not allow this approach to scale.

Bernstein et al. explore theoretically middle-tier caching in [BFG+06]. The authors propose an extended serializability model for data consistency in which inconsistency is bounded to miss a maximum number of update transactions (termed freshness). This consistency is very relaxed and contrasts sharply with the strong consistency provided by our approach. On top of a master database (again a single point of failure), they simulate replicated caches (each one with a cache manager) that may hold replicas of the data items contained in the database. There is a special cache manager called master manager. Read operations can be done by the master or by the standard cache managers. Write operations are processed only by the master manager. After a transaction commits, the master manager sends the changes to the caches that hold the modified items. The simulation performed in the paper is evaluated with an ad-hoc benchmark. The scale out is measured injecting a percentage of 70% of read transactions. The results show that the system is able to scale out up to five cache replicas. Our approach of Chapter 4 is a real implementation evaluated with an industrial benchmark, allowing to scale out a cluster up to ten replicas and providing a high level of consistency via snapshot isolation.

6.1.1.3. Recovery

One of the first works that introduce online recovery in a replicated system is [CL00]. The paper addresses a proactive recovery protocol that reduces the window of vulnerability of a secure replicated system. In proactive recovery, replicas are restarted through a hardware timer and they reboot from read only memory. In this way, any effect from a successful attack to the replicas is destroyed. When a replica restarts, it takes the state from the other working replicas using an agreement protocol to guarantee that an attacked replica does not corrupt the state of the recovering replica. Despite the main focus of this system is on security, it has a similar goal to our online recovery protocols in the sense that they aim to provide continuous availability minimizing the impact (overhead) in the performance of the cluster.

The literature on database management systems (DBMSs) describing how to perform data recovery centralized database systems is plentiful [BHG87, GR93, BN96]. Recovery in centralized DBMSs tries to minimize human intervention but requires to stop the processing of new client requests whilst the recovery process is being done (offline recovery). Offline recovery is the only option for architectures based on a centralized back-end (multi-tier or not). For this reason, many efforts have been devoted to design recovery protocols that are as fast and efficient as possible [MHL+92]. However, the high availability requirements of current multi-tier applications have turned offline recovery into a major problem to solve, not only in the back-en tier but also in the middle tier. In the cluster architectures that support these applications, the system as a whole must continue processing requests even in the advent of replica failures. Thus, online recovery is mandatory to preserve high availability of applications whilst failed or new replicas are re-joined to the cluster. Online recovery also allows to scale out a cluster adding more replicas if the underlying coordination protocols are designed carefully.

In the context of database replication, Kemme et al. present in [KBB01] an online recovery

protocol based on locking. The basic idea is that when a recovering replica joins a group of working replicas, one of the working replicas stops processing and then sets recovery locks over the whole database. This replica takes care of transferring the necessary data to the recovering replica. During the recovery process tuples are read one by one. Upon reading a tuple the recovery lock is removed and its value sent to the recovering replica. Transactions arriving to this temporary offline replica and to the recovering replica during the recovery phase are queued until the recovery phase terminates. Once the recovering replica is up-to-date, the offline and the recovering replicas start to process the enqueued transactions until they catch up with the rest of the working in the system. Then, they resume the normal processing of requests. The recovery protocol is based on an extended version of a GCS abstraction called *enriched view synchrony* (EVS) [BBD97] that is used to deal with failures. This work has several differences with regard our recovery protocol in Chapter 4. The first one is the context. Our work addresses replicated multi-tier environments using vertical replication, being the back-end tier recovered at the same time as the middle tier. The second difference is that the protocol of Kemme et al. allows the system to remain online by assigning the recovery tasks to one replica, that becomes temporary offline. In our protocol, the recovery process is distributed among all the working replicas of the cluster. In the same way as in the protocol of Kemme et al., transactions that arrive to the recovering replica are queued until the recovery phase is finished. Once the new replica is recovered, it starts processing the queued transactions and finally starts processing new requests. Finally, we include a middle tier log that allows fast recovery of the whole replica state (middle and back-end state) when the replicas fail for short periods.

In [JPA02] is presented a protocol for replicated databases that overcomes the drawbacks presented in the work of Kemme et al. First of all, the protocol is not intrusive, that is, it does not require to access the source code of the database to be implemented. It works at a middleware level, outside the boundaries of the DBMS. This makes the solution applicable to a wide range of scenarios: transactional file systems, transactional distributed objects... The protocol aims to provide continuous availability using a log-based approach. Continuous availability cannot by provided by a locking-based recovery protocol, since at the beginning all update operations are blocked on the recovery locks what results in unavailability of update requests. In addition, the protocol provides a set of optimizations that enables to deal efficiently with simultaneous and cascading recoveries minimizing redundancies in the recovery process and therefore saving a significant amount of processing capacity. Moreover, the protocol allows multiple replicas to collaborate in the recovery process. Finally, the protocol is very flexible, enabling to devote a wide range of resources to recovery enabling autonomic adaptation. The recovery protocol presented in Chapter 4 uses this work as a reference to build a high available and scalable multi-tier infrastructure.

In [AMD+06], another recovery protocol for replicated databases is described. It also uses a middleware to extract the protocol from the DBMS. The protocol uses only one recoverer to synchronize the state of the joining nodes. Upon a failure is detected in a node, the protocol selects a recoverer that starts the creation of partitions (*DB-partitions*) managed by special *recovery transactions*[2]. These DB-partitions group the missed updates for each view. The number of partitions to sent depends on the number of views missed by the recovering node.

[2]To do this, the protocol modifies the database schema by adding a table containing, for each view, the crashed nodes and the set of updated data items. It keeps updated this table using the strong view synchrony facility of GCS and the changes received through reliable multicast.

The recoverer node sends to the recovering the missed updates in each required partition in the context of a recovery transaction. When the recoverer has sent all the data items of a given partition, the recovery transaction finishes (releasing the corresponding partition), even if the recovering has not finished its recovery process. User transactions that contain update conflicts with transactions stored in DB-partitions that have not been sent yet to the recovering node are blocked whilst the recoverer sends the partitions.

In [LK08], the authors explore two online recovery mechanisms for replicated databases that are similar to those presented in Chapter 4. The first one allows to transfer the updates that the joining node has missed during its downtime and the second one allows to transfer the complete state of the database to a new node. As in our proposal, the recovery protocol has a mechanism that decides when to transfer the complete state of the database or just the set of committed changes missed by the joining node. However, in this approach only one active node acts as recoverer and requires human intervention to handle part of the recovery process. Moreover, each time that the joining node receives and processes the set of changes sent by the recoverer, it notifies the recoverer to send more changes, what implies to exchange more messages between the parties. Our protocol also solves the synchronization issues that arise in multi-tier architectures between the middle and the back-end tiers.

As it happens with replication, recovery in multi-tier architectures has not been widely studied until the beginning of this decade. In [WV01] it is described a protocol for the recovery of stateful applications that is applied to non-replicated multi-tier system architectures. The protocol is coined *server reply logging method* and assumes that the applications to be recovered are piecewise deterministic. The recovery process relies on message logging in order to reconstruct the original messages when an application process fails. Log files can be a trade off for certain critical applications (such those that require real-time), but in general it is affordable for business-oriented applications. The state of the processes is saved periodically to stable storage, what is called *installation point* (IP)[3]. The recovery execution after a failure is resumed from the most recently IP. The protocol distinguishes two roles with regard to processes: client and server. For each client process and for each client request, the server saves log entries with the responses, before sending each reply to the client. This way, the server is able to return already processed requests to failed clients that are being recovered. When the server process is recovered after a failure, it reconstructs an in-memory table with the replies of the clients from the last IP. In order to not overflow the server log, the clients inform the server when log entries are no longer needed.

Roger Barga et al. extend the ideas of the recovery protocol described above and implement it in multi-tier architectures in the context of the Phoenix project [Micd, BLW02, BLSW04]. The aim of the project is to increase the availability of applications and avoid to the user the operational tasks of dealing with errors. Their system is based in the COM+ component model from Microsoft. It provides a framework that provides recovery guarantees for multi-tier applications using interaction contracts between the different components involved in each user request. It also provides exactly-once semantics for user invocations, masking the possible failures to the final users. Recovery is based on deterministic replay of messages on the deterministic components that make up the applications (clients, application servers, data servers and users). Three types of components are considered: (1) *Persistent components* - Their state should persist across failures. Phoenix logs component interactions and, if a failure occurs, automatically recovers all persistent components up to the last logged interaction; (2)

[3] IPs are similar to checkpoints in database terminology.

Transactional components (e.g. data servers) - These components are recovered up to the last successfully completed transaction. The running transactions during a failure are aborted by the database. It is supposed that the application is written to deal with (retry) failed transactions; (3) *External components* - They are used to model human users and thus can not be recovered. To avoid non-determinism, each component is able to checkpoint its state when required in a log file –what is also called installation point (IP)– through a recovery manager. When the system has to be recovered, the recovery manager scans the log. A component is recovered from its last IP. The required information is reconstructed from the log entries and injected to the component. This is ensured by means of the contracts that specify the behaviour of each pair of components in the presence of failures.

However, on the contrary to the recovery approach presented in this thesis, these two works on multi-tier recovery have been focused in the context of centralized architectures, and therefore they do not perform online recovery.

Some work on how to perform recovery in a replicated cluster has been also done in the context of FT-CORBA. In [NMM01], it is addressed how to perform recovery of stateful objects in Eternal, the transparent replication system for CORBA. Depending on the replication schema selected (active or passive), different mechanisms are used. In active replication, the failure of a single replica is easy to mask because the other replicas keep a consistent copy of the object. When recovering a failed replica, the state of the objects must be synchronized with the consistent state of an existing operational replica (that acts as a recoverer). The recoverer sends the messages lost by the recovering replica until its state is updated. A checkpointing mechanism is used to send the right messages to the recovering replica. In passive replication (primary-backup), if a backup replica fails, it is simply removed. If the primary replica fails, one of the backups is elected to be the new primary. Before the new primary replica can start processing new normal invocations, its state must be synchronized with the state that the old primary replica had just before it failed using a log file. In both replication styles, in order to synchronize the state of the objects in new or recovering replicas, the recovery protocol requires that the system be in quiescent state in order to start the recovery phase and remain in quiescent state during the whole process. While this can be acceptable in a non-transactional system, in a transactional system is not. The reason is that reaching a quiescent state in a transactional system implies to wait until all active transactions at the time of recovery have finished and to delay the processing of new requests until the recovery is completed (the state of a transactional system can be fairly large). Moreover, the paper only considers the recovery of in-memory state and they do not describe how to achieve consistency along the different tiers when recovering a multi-tier application.

In order to provide complete high availability to the replicated cluster infrastructure described in [WKM04], the authors also describe a recovery protocol that allows to incorporate/recover application server replicas. This protocol is similar to our recovery protocol described in Chapter 3. However, as their cluster infrastructure shares the database, they do not need to transfer its state to the other replicas.

Another initiative dealing with recovery is the *recovery oriented computing* (ROC) project [Ber, BP01, OGP03]. ROC is a joint Berkeley/Stanford project that emphasizes recovery from failures rather than failure-avoidance. The project assumes that systems are not free from failing and recognizes redundancy as a means to provide continuous service delivery while certain portions of the system are isolated due to failures. In a similar way than Castro and Liskov [CL00], the project also emphasizes proactive restart of system components. Currently,

most systems that support Internet services (e.g. the Apache HTTP server [Apab]) already perform proactive restarts of components before they fail in order to improve their overall availability. ROC shares the objective of increasing the system availability with our replication and recovery protocols. However, the way they aim to guarantee availability through recovery is different than ours. They focus on recover components in advance (e.g. repairing latent errors) to not to propagate the errors to other parts of the system. Our approach, in contrast, focuses on how to perform recovery once the failures have appeared and without disrupting regular processing of the rest of the system. However, both approaches are complementary. ROC could be applied within the components of each in the vertical replicas (web server, several application server parts, DBMS...) to try to avoid potential crash failures.

A more recent approach deals with recovery in a WAN data warehousing environment [LM06]. The paper presents HARBOR, a system used to recover what authors call "updatable warehouses", a new kind of warehouse-like system used in modern CRMs and data mining applications that support fine-granularity insertions of new data and occasional updates of incorrect or missing historical data. Their approach leverages data redundancy in remote nodes that enables a recovering node to determine which tuples need to be copied or updated and query working nodes to recover an up-to-date snapshot of the database. Their approach is similar to ours in the goal of enabling the recovery of a node without disrupting other working nodes. However, their approach is different from ours in two main aspects; firstly, the execution environment. They focus on SOA in a WAN distributed environment, whilst our work is addressed to clusters in LAN environments where each replica runs a multi-tier infrastructure. Secondly, their aim is not to provide continuous availability for the whole system but to exploit remote data redundancy in the warehouse to enable recovery of failed nodes.

Buzato et al. describe in [BVZ09] an analysis of how crashes and recoveries affect the performance of a replicated dynamic content web application. The work of the authors is based on Treplica, a middleware for building dependable applications based on replication and recovery. Treplica is based on state machines and asynchronous persistent queues implemented using the Paxos [Lam98] and Fast Paxos [Lam06] algorithms. The queues guarantee the total order and provide the required state persistence for the replicas. Persistence means that a replica bound to a queue can crash, recover and bind again to its queue. The queue ensures that its state is preserved, that is, no enqueues from other active replicas have been missed. Application programmers use the state machine interface to perform the required operations on Treplica (e.g. manage events, conditions and actions). This means that dependability can not be provided transparently to applications because the programmers have to introduce the required calls to the API provided by the replication middleware. The recovery process is as follows. When a replica crashes and triggers the recovery process, a stateless instance of the application and an asynchronous queue are created and associated. It is responsibility of the queue to provide the recovering replica with the state which it must be reset, in the form of a locally obtained checkpoint and an associated suffix of the queue's history. The required suffix to complete the synchronization is learned from the active replicas using Paxos. The evaluation of the recovery process was done using a modified version of the TPC-W benchmark [Tra05]. However, the resulting application does not rely on a database and the whole application state resides in memory.

6.1.1.4. Advanced Transaction Models

With regard advanced transaction models, they were proposed during the eighties in the database community to avoid limitations of the traditional flat transaction model. The first advanced transaction models studied in deep were nested transactions [Mos81], long running activities [Gra81] and SAGAS [GS87] described in Chapter 2. During the nineties, other advanced transaction models were widely studied in database literature [Elm92, JK97]. Most of them address the requirements of specific application domains (e.g. collaborative work, 3D design environments, workflows...) [AAE+96, WS97]. In [CR90] it is proposed a theoretical framework to model the atomicity and isolation properties of transaction models.

In [BLBA00] the authors propose ODBC support for highly available transactions in the presence of short-lived database server failures. They target, as we do with the third protocol in Chapter 3, an scenario with transactional long running activities (such as OLAP) in which a crash is more likely to happen (due to the long duration of transactions). In this scenario the amount of lost work may be very high and applications may require operator-assisted restart. Their approach consists in making the database connection state persistent, so it can be recovered upon a crash of the database server. Upon restart of the database server, the ODBC driver reconnects with the DB server and the DB server recreates the connection state. Therefore, database crashes are masked transparently to client applications. Our approach attains these properties in the context of a J(2)EE multi-tier architecture. Instead of using persistent sessions, we resort to replication and end-to-end reliability based on exactly-once transactions to attain transaction availability and transparent failure masking.

6.1.2. Industrial Work

Nowadays, the industry is dedicating a lot of effort on solving fault-tolerance problems to provide high availability to enterprise applications. A lot of efforts are being done in J(2)EE application severs. Lightweight containers based on Java (e.g. Spring [Int], PicoContainer [Lig], HiveMind [Apah]...) can also profit from these efforts[4].

Scalability is the other major objective to achieve by the industry in both, application servers and lightweight containers. The current middleware solutions that aim to provide scalability for applications are based on the use of distributed caches.

6.1.2.1. Application Servers

This section analyzes the clustering features related to high availability and scalability that –at the time of writing this thesis– provide the most important application servers. Almost all of them are J(2)EE based. Two main groups can be distinguished: proprietary and open-source application servers. On the proprietary side, with regard to J(2)EE, the most important ones are: Oracle Application Server [Oraa], IBM Web Sphere [IBM] and BEA Web Logic [BEA]. Finally, Microsoft Application Server is not based on J(2)EE technology. On the open-source side, J(2)EE is the only alternative. JBoss [Reda] and JOnAS [Objf] have been the alternatives for years. However, the great interest of the industry in the J(2)EE application

[4]These frameworks have gained positions during the last few years as an alternative to the complexity of J(2)EE application servers. They are based on the notion of dependency injection (Also known as inversion of control or IoC [Fow04]) and are used in the development of small and medium-size applications that do not require all the services provided by an application server.

server market has caused that the Apache Foundation –very important in the open-source community– and even SUN Microsystems provide their own open-source alternatives named Geronimo [Apag] and GlassFish [Suna] respectively. Each application server vendor differs in the way that understands high availability and scalability and the consistency that is required by applications behind the scenes.

All application servers described below provide basic features for implementing clusters that are related to high availability and scalability, such as load balancing mechanisms or session affinity (See Chapter 1, Section 1.4). However, state consistency, high availability and scalability can not be achieved in a cluster by simply combining these features. Moreover, all the solutions offered by the vendors are based on the horizontal replication approach at the application server level, so the single instance of the database is a potential point of failure for the cluster. Additionally, the state synchronization of the persistent components in the different replicas of the application server is done through the shared database, what implies a loss in performance due to the unnecessary accesses to the database.

6.1.2.1.1. Oracle Application Server

The J(2)EE application server from Oracle is called Oracle Application Server (OAS)[5] [Oraa]. Oracle claims that all the OAS components in a cluster (load balancers, HTTP servers, J(2)EE containers, data sources...) can be deployed in a redundant fashion in order to provide high availability [Ora06].

An OAS cluster can be configured following a primary-backup (active-passive) or an update-everywhere (active-active) model. A priori, the update-everywhere model offers more scalability facilities and transparency to the user. OAS uses its own replication framework to replicate stateful session beans (SFSBs). The framework uses a reliable messaging service to guarantee that the state is delivered to the replicas of the cluster. OAS allows to specify the number of destination replicas and if either all attributes of a SFSB should be replicated or only those that have been changed. There are two replication policies that decide when to replicate the SFSBs: (1) *On Request End* - The SFSB state is replicated to all replicas at the end of the EJB method call. This option is reliable, but does not provide very good performance; (2) *On Shutdown* - The SFSB state is replicated to only one replica when the JVM is stopped. This option is not reliable because the node holding the JVM can terminate unexpectedly. In contrast with the replication protocols presented in this thesis, none of these replication policies is transaction aware.

Sessions can also be persisted in a database for those applications that are critical enough. Session persistence can be done synchronously or asynchronously with regard to the client requests. Of course, this option can degrade the performance of the applications and turns the database where the state is saved into a single point of failure.

With regard to entity beans (EB), OAS does not provide any consistency guarantee for the EB instances in the replicas when using and update-everywhere approach. This means that application state (EB state) can not be cached at application server level when running in an OAS cluster. This implies to reload the state of every EB from the database each time that it is accessed by the application. However, Oracle has recently bough Tangosol Coherence [Orab], an in-memory cache mechanism that allows replication and can be integrated with application

[5] OAS is the result of the fusion of the Ironflare's Orion [Iro] J(2)EE application server and the original application server from Oracle.

servers or lightweight containers. It is supposed that, in the near future, Coherence will be integrated in OAS. More information about Coherence can be found in Section 6.1.2.2.1.

OAS also provides a tool to backup and restore operations called Recovery Manager (RM). However, this solution does not provide on-line recovery for the application server state because it still requires manual operation to guarantee the consistent state of the recovering replica after the recovery process.

6.1.2.1.2. BEA Web Logic

WebLogic [BEA] is J(2)EE application server developed by BEA. The clustering and high availability features of WebLogic are similar to the ones provided by its competitors [BEA07]. With regard to EJBs, WebLogic provides state replication for stateful session beans (SFSBs) using replica-aware stubs. As the stubs used in our protocols, these stubs contain extra-information to locate, create and access instances of SFSBs in the cluster replicas following a primary-backup approach. SFSB state replication is not transaction aware, and it is done at the end of every method invocation on the SFSB.

When the primary replica for a client fails, the cluster-aware stub detects the failure and redirects next requests to the backup replica. When receiving a failover request, the backup replica creates a new SFSB instance using the replicated state data and continues processing the request. In addition, the WebLogic replica that receives the failover request, chooses a new backup replica (if available) to hold the replicated session state data. The client stub is updated with the information about the new backup replica when the response is sent back.

Multiple instances of a particular entity bean (EB) may coexist in a cluster. This means that consistency must be guaranteed for the EBs state used by the application. When transactions are used to achieve consistency, WebLogic provides an optimistic concurrency control mechanism based on versions to keep consistent the state of the EB instances in all the replicas. In a cluster, when an EB using such mechanism is updated, notifications are broadcast to other replicas at the end of each transaction. However, no additional mechanisms to detect possible concurrent conflicts in EBs on different replicas are provided at the applications server. The shared database acts as the element that guarantees consistency in those scenarios. In addition, the version information is stored in columns added to the database tables, what implies a loss in performance checking additional information against the database when EB modifications are performed.

6.1.2.1.3. Geronimo/IBM Web Sphere

IBM has recently split its WebSphere Application Server (WAS) [IBM] in two branches: the first one offers a community edition of WAS based in open-source; the second one offers a commercial version of the community edition with extended features. The WAS Community Edition is built on top of Geronimo. Geronimo [Apag] is the latest important effort in the open-source community to build a J(2)EE compliant application server. Geronimo was born as a project of the Apache Foundation (funded by IBM). Geronimo project is quite recent and the application server core was designed from the scratch. Nowadays, the efforts are centered on providing and testing the basic services for a J(2)EE compliant application server, so the current version does not include advanced features for clustering related to high availability and scalability. These features seem to be planned for future versions (e.g. Terracotta [Ter]

or GCache [Apaf] are planned to be integrated as second-level caches [Apae]. For additional information on second-level caches see Section 6.1.2.2).

However, the commercial version of WAS offers clustering features related to high availability[6] [IBM05, IBM06]. WAS does not support the instantiation of a specific instance of a SFSB in several replicas. However, it provides a service called Data Replication Service (DRS) that allows to replicate stateful session beans (SFSBs) in memory in order to survive replica failures. The DRS uses a reliable multicast service to propagate the state information to other replicas. WAS is transaction aware, that is, SFSBs accessed inside a transaction are replicated when the transaction completes.

Failover is performed by the Workload Manager (WLM) service. The WLM redirects the client requests to another replica if the one that was assigned is unable to process them. However, if the replica fails while processing a request, the WLM cannot fail over to another server. In this case, the application is notified with an exception. This means that WAS does not provide highly available transactions.

With regard to EBs, their state can not be replicated. The state of the EB instances can be cached between transactions in an application server replica. However, as a last resort, the state consistency in the same EBs instances located in different replicas of the cluster must be guaranteed by the shared database.

IBM has introduced in early 2008 WebSphere eXtreme Scale (WXS)[7] in its suite of high quality application infrastructure products [IBM08]. It is the commercial response of IBM to the eXtreme Transaction Processing (XTP) concept [PGNS07]. WXS claims to offer business applications an extremely efficient and scalable processing for large volumes of transactions. However, there are no details available about its implementation.

6.1.2.1.4. JOnAS

JOnAS [Objf] is the open-source application server of the Objectweb community[8]. From version 4.7 JOnAS offers transaction-aware replication protocol to provide high availability for stateful session beans (SFSBs) [Objd]. The entity beans (EBs) are not considered by this protocol. The protocol uses a primary-backup approach, but clients can connect with any of the application servers available (no session affinity). In order to ensure consistency, the protocol uses eager replication. So, the changes on SFSBs are replicated in a single message before committing each transaction. In order to multicast the changes to the replicas of the cluster the JGroups [JGr] group communication system is used. Moreover, the protocol guarantees exactly-once execution of client requests. In order to avoid the re-execution of requests, the protocol logs the requests and their corresponding responses. As the primary can fail after sending the replication message but before committing the transaction, the primary introduces a "marker" in the database as part of the transaction. During failover, if the primary fails, the new primary can query the database to check if the marker is there for each message received. If the marker is not found, the request is re-executed in the new primary. In order to improve the high availability of the solution, each application server instance can be attached to a local database to store the markers. These databases must run in different processes than the database used by the application. So, the failure of one of these databases for markers

[6] High availability features are provided only in the Network Deployment distribution.
[7] Formerly known as Data Grid.
[8] BULL maintains a commercial version offering additional commercial support and features.

does not affect the other application server replicas. The database used by the application, as it happens in other approaches, must be shared among the application server replicas of the cluster (single point of failure).

Our replication and recovery protocols presented in Chapter 4 extend this protocol providing not only high availability for SFSBs but also high availability for EBs and scalability for the applications deployed in the cluster.

6.1.2.1.5. JBoss

JBoss [Reda] is the most extended and mature open-source J(2)EE application server. JBoss is one of the first application servers that provided clustering facilities and high availability for application server state. In 3.x versions, JBoss provides replication only for stateful session beans (SFSBs) [LBG04]. The replication process is not aware of transactions and the state of SFSBs is multicast to the rest of the replicas after each method invocation. For this purpose, JBoss uses the facilities provided by the JGroups [JGr] group communication system. JBoss sessions can be configured either without session affinity, so that each client request is executed in a different JBoss instance (in a round-robin fashion) or with sticky sessions, behaving as a primary-backup approach. The exactly-once execution of client requests is not ensured by the replication protocol. With regard to the database in which to store persistent data is always shared among all JBoss replicas. Therefore, the database becomes also a single point of failure in this approach.

4.x versions have improved the clustering facilities extending high availability to entity beans (EBs) [Red06]. High availability for EBs is achieved by means of JBossCache [Redb]. JBossCache is discussed in Section 6.1.2.2.2.

6.1.2.1.6. GlassFish/SUN Application Server

GlassFish [Suna] is the open-source version of the SUN Java System Application Server (SJSAS). It is maintained by the GlassFish open-source community created by SUN in 2005[9]. With regard to the clustering features for high availability, they are still a work in progress. The core element for the clustering capabilities is the Group Management Service (GMS)[10]. The GMS is a clustering framework that includes a group communication system and will allow to build fault tolerant, reliable and available systems. Currently, only high availability for session state is provided. It can be achieved using in-memory replication to other replicas in the cluster or using a shared database. The second solution has been the traditional solution on SJSAS to provide session availability. However, this solution is difficult to implement and maintain because it involves a database. Moreover if the database is not replicated, it could be a single point of failure for session information. In-memory replication in GlassFish is a similar to the one commented for JBoss. It requires the GMS and uses the memory on other replicas to maintain a copy of the HTTP and SFSB state. Cluster replicas are organized following a ring topology. Each replica of the cluster replicates session-state changes to the following replica

[9] SUN has copied the same dual-offering business model that Red Hat and IBM offers with JBoss and Geronimo respectively. They offer a free open-source edition with limited features to use-at-your-own-risk and a supported commercial version.

[10] Also know as Shoal.

(partner) in the ring. When a replica fails, the GMS is able to reconfigure the ring structure selecting new partners for the affected replicas.

6.1.2.1.7. Microsoft Application Server

The application server from Microsoft is called Microsoft Windows Server (MWS) [Micb]. MWS is not a J(2)EE compliant application server. In the Microsoft's approach, the application server facilities are embedded in the operating system. The application server technologies embedded in MWS include a Web engine (Internet Information Service, IIS), a framework to develop applications (.NET), a security system to enable applications the access to resources and a transaction manager. This technologies are complemented by the other elements included in the Windows platform (active directory, event log, message queues, load balancer...). MWS also includes high-available solutions for the applications running in clusters. It provides two clustering technologies called *network load balancing* (NLB) and *server clusters*. The NLB is intended only for stateless applications. Server clusters [Mic03] are used for stateful applications. Up to eight servers can integrate a MWS cluster.

MSW server clusters do not include an automatic load balancing mechanism for applications. However, applications can be moved manually around the different replicas of the cluster in order to distribute the load.

Failover can be done at replica level or at application level. At replica level, when the cluster monitoring mechanism detects a failure of a replica, it re-starts automatically the failed server's workload on one or more remaining replicas. If a single application crashes, MWS tries to restart the application on the same replica. If this fails, the application's resources are moved to another replica and the application is started again. This behaviour can be changed by modifying the recovery policies. For example, the recovery policies allow to specify if an application should be re-started on the same server or if the workload should be automatically the re-balanced when a failed replica comes back online. This way, client re-connections can be made transparent if application is restarted at the same node, because the applications, file shares... can be restarted at exactly the same IP address. When applications restart after a failure, MWS does not restore the in-memory state used by the application before the failure (The in-memory state is neither stored nor replicated). Only the data written out to the persistent storage is available to the application. So, if the client uses stateless connections, failover is transparent if the failure occurs between requests. Otherwise, the client receives a notification to help him/her to re-establish the contact with a replica.

MWS server clusters provide facilities to partition the data used by an application. For example, a SQL server database can be scaled-out by partitioning the database in several pieces and using database views to provide transparency to the applications that use it. Each part can be failed over independently within the set of replicas so that in the event of failure, a partition of the application remains available.

6.1.2.2. Middleware Caches

The default cache mechanisms provided by application servers for stateful components usually provide the ANSI isolation levels found in databases (read uncommitted, read committed, repeatable read, serializable) to guarantee the correct execution of concurrent transactions in a single application server. However, none of them provide a cache with snapshot isolation

(SI). SI guarantees almost the same consistency as serializable allowing a better degree of concurrency. Moreover, a SI cache at the application server would work in perfect synchrony with the SI level of the most popular and efficient DBMSs (Oracle, Microsoft SQL Server, PostgreSQL...)[11]. Another drawback of the standard caches at the application server is that they are not prepared to guarantee the consistency of stateful components among several replicas in a cluster configuration.

This section presents several caching solutions suitable for being integrated as second-level caches in application servers. They provide facilities (e.g. replication) that can help to provide high availability and/or scalability to application server clusters. This kind of caches should keep consistent the stateful components that are accessed in the replicas of a cluster providing at the same time good performance for both, read and write operations. Moreover, when a stateful component is persisted, the cache and the underlying storage should be kept properly synchronized to guarantee the consistency in the multi-tier architecture. If these two layers are synchronized, it is easier for example to build a recovery mechanism.

The transaction-aware cache presented in Chapter 4 uses versioning of components to provide SI guarantees in a single instance of the application server or in a cluster. SI allows to increase the degree of concurrency because read-write conflicts are avoided and write-write conflicts between two transactions are solved aborting only one of them. Moreover, it guarantees that the cache is consistent with the data stored in the underlying database.

6.1.2.2.1. Oracle-Tangosol Coherence

Coherence [Orab] is an in-memory cache based on JCache[12][Sune]. Coherence allows to configure a wide set of parameters in order to build the right cache architecture for each application. For example, it allows to configure different topologies depending on where the data resides and how is accessed. However, the use of the cache for applications is transparent at cluster level. This means that, independently of the cache architecture configured, each participant in the cluster has the same logical view of the data and uses the same interface to access them. In the context of this thesis, it is important to study the combination of replication, transactions and the cache synchronization with data sources provided by Coherence.

In order to configure how the cache works, four different options are provided: (1) *Replicated* - This approach replicates data to every member in the cluster. This allows great performance in read operations, because every member has the data locally. However, this fact is also responsible of the poor scalability for write operations, because every member has to update each modified data. In order to perform consistent write operations a concurrency control mechanism is used; (2) *Optimistic* - It is similar to the previous approach but without using the concurrency control mechanism. It offers great performance but it can produce data inconsistencies; (3) *Distributed (Partitioned)* - In this approach, data is not stored in all the members of the cluster. The user can select the number of nodes where to store each piece of data; (4) *Near* - It is an hybrid configuration. It offers a front view cache (local cache) representing all the data accessed locally and that is stored in another centralized or multi-tiered cache. The front view is configurable to invalidate entries using different strategies.

[11] Currently, in these DBMSs, the serializable isolation level does not provide a real serialization of transactions. In order to improve the performance, serializable is implemented to provide SI guarantees.

[12] JCache was the failed effort at Java Community Process level to define an API to provide semantics for temporary, in memory caching of Java objects across multiple JVMs.

In order to manage the concurrent accesses to data, Coherence provides several options including explicit locking and lock-free programming models and the integration with J(2)EE application server by means of JTA transactions. To deal with transactions, Coherence provides a thread-safe map structure that maintains two copies of the data: base (committed) and local (uncommitted). Read operations use the local map first and if the elements are not found, the base map is used. Write operations always use the local map (using a mark to denote delete operations). In order to use consistently this structure it is necessary to set the concurrency (pessimistic and optimistic) and isolation levels (GET_COMMITTED, REPEATABLE_GET, SERIALIZABLE). The advantages of snapshot isolation offered by our cache in Chapter 4 are not provided by Coherence. The possible combinations of both parameters are shown in table 6.1.

	Pessimistic	**Optimistic**
GET_COMMITTED	Write operations lock the corresponding resources on the base map.	No additional processing.
REPEATABLE_GET	Same as GET_COMMITTED plus read operations lock the corresponding resources on the base map.	Read operations move the retrieved values into the local map.
SERIALIZABLE	Same as REPEATABLE_GET plus read operations on a set of keys lock the entire base map to prevent it from adding or removing resources.	Same as REPEATABLE_GET plus read operations on a set of keys move all the values to the local map and disconnect the base to prevent phantom reads.

Table 6.1: Isolation semantics in Coherence transactions

In replicated caches, the changes on the state of objects can be done in any of the cache replicas (update-everywhere approach). An object is replicated to other caches either the first time is put in the cache or when it has been modified in the context of a transaction. So, inside a transactional context the changes are replicated at the end of the transaction to avoid network overhead. The protocols presented in chapters 3 and 4 also avoid the network overhead because changes are multicast only when the transaction is about to commit and no conflicts have been detected with other local transactions.

In order to commit the changes in transactions, Coherence uses a 2PC protocol combined with the concurrency and isolation levels selected previously. When using pessimistic concurrency control, the prepare phase checks the locks for all resources contained in the local map. If this succeeds, the commit phase copies the local data in the base map and the resources are unlocked. If any exception occurs in the prepare or commit phase, the local map is cleared and the resources are unlocked. With an optimistic concurrency control, the prepare phase attempts to lock the resources contained in the local map. If locks are granted, the key sets used are validated using a validation entity. This validation entity allows to define strategies to verify the concurrent execution of transactions. Finally, the commit phase copies the local data in the base map and the resources are unlocked. If any exception occurs in the prepare or commit phase, the local map is cleared and the resources are unlocked. Thus, in order to guarantee consistency in replicated caches a considerable amount of time must be spent when using a 2PC protocol combined with locks.

Some other issues may arise using replicated caches with Coherence. For example, if two concurrent transactions are executed either in the same or in two different replicas and modify

the same elements A and B but in different order, a deadlock can happen when locks are trying to be obtained in the prepare phase. Then, both transactions can be aborted after they have replicated their changes on elements A and B. This contrast with our approach based on snapshot isolation shown in Chapter 4 where only one transaction should be aborted. Moreover, our approach is able to detect conflicts locally (within the same replica). In this case, one of the transactions is aborted and does not replicate its changes.

In order to deal with the underlying persistent storage, it provides a component called *CacheStore*. A CacheStore allows synchronization with any data source, including databases, web services, filesystems... For example, the CacheStore can be an object/relational mapper (O/R mapper) when dealing with a relational database. The documentation does not state clearly if in a cluster configuration, each instance of the cache can be connected to a particular instance of a data source. For example, this would be useful in order to configure a vertical replication approach. As usual, if only one instance of the data source can be used, it becomes a single point of failure for the cluster. Coherence offers four policies to synchronize the cache with the data source: (1) *Read-Through Caching* - This policy delegates to the CacheStore to load elements that are not in the cache when they are requested by applications; (2) *Write-Through Caching* - When a piece of data is updated, the cache will not return the control to the application until the data is not stored in the underlying data source (synchronous write). The caches used in chapters 3 and 4 always use synchronous write operations with the underlying database to guarantee the recovery process of the replicas; (3) *Refresh-Ahead* - This policy allows to refresh frequently accessed entries only when their expiration time from cache is going to finish; (4) *Write-Behind* - Modified cache entries are asynchronously written to the data source after a certain amount of time. When modified, entries are placed in a queue and written by other thread when the delay expires.

When Coherence is configured for using transactions and combined with an O/R mapper (e.g. Hibernate), it provides versions for persistent components. However, unlike our cache, the transactional semantics provided by the resulting combination are read committed.

6.1.2.2.2. JBossCache

JBoss Cache [Redb] is a second-level cache developed in the context of the JBoss application server. As Coherence, JBoss Cache is highly configurable, transaction-aware and can be deployed in a cluster. JBoss Cache is based on a tree-like structure. Each leaf of the tree represents a cached element and the root and the intermediate nodes allow to classify the elements in the cache. For example, if the row $R1$ of table $T1$ from database DB_A is cached, the view of the tree is $/DB\_A/T1/R1$. A JBoss Cache instance can have multiple roots, allowing several trees in a single cache instance. As it is also a transactional cache, application interactions with the cache can preserve the ACID properties when transactions are used. JBoss Cache transactions support the ANSI isolation levels for dealing with concurrent transactions. Finally, it also provides a mechanism, called *CacheLoader*, that allows to synchronize the state of the elements of the cache with (persistent) data sources. By default though, all write operations in a CacheLoader are synchronous.

JBoss Cache provides two ways to maintain data consistency in the cluster: *replication* and *invalidation*. With replication, the changes in one replica are replicated to all replicas in the cluster at the end of a transaction. When a transaction is committed, its changes are applied in all replicas. In the current implementation, everything that has been put inside the

cache is replicated, including read-only operations as (e.g. the results of a SELECT query). In order to replicate the changes, the JGroups GCS is used. If invalidation is used, the changes on the cached elements are not replicated. Instead, the elements modified in the cache are tagged as out-of-date at the end of a transaction. When the transaction commits, all tagged beans will be evicted from all the caches in the cluster. This approach minimizes the amount of information that is multicast to the replicas, as invalidation messages are very small. On the other hand, all tagged data should be reloaded from the database the next time that is accessed.

In a cluster of caches, a 2PC protocol is used to guarantee consistency. Upon committing a transaction, JBoss Cache multicasts a prepare call, which carries all modifications relevant to the transaction. The remote caches apply the modifications if no conflicts are detected. Once all remote caches respond to the prepare call, the originator of the transaction broadcasts a commit. This instructs all remote caches to commit their data. If any of the caches fail to respond to the prepare phase, the originator broadcasts a rollback.

In the first versions of JBoss Cache, concurrency control can use a pessimistic or an optimistic locking approach. The pessimistic approach uses a lock per tree node. Before performing any operation in a node, the readers must obtain non-exclusive read locks and writers must obtain exclusive write locks. The locks support lock upgrading (from read to write) within the scope of a transaction. This approach do not presents several drawbacks. First of all, concurrency is reduced since a read lock prevents a writer to obtain a write lock. This also introduces the possibility of deadlocks. JBoss Cache includes an optional deadlock detector, but it increments the processing time for transactions. Moreover, in order to solve deadlocks, the mechanism must abort all the conflicting transactions because it can not decide which transaction to commit. The 2PC protocol combined with locks makes the commit procedure too heavy-weight to be used in a cluster [GHOS96]. Thus, with this approach JBoss Cache can provide high availability for the cached data, but it can not provide scalability.

To overcome the deadlock potential, JBoss Cache also offers an optimistic locking scheme that uses data versioning on each node. It copied any nodes accessed into a transaction workspace, and allowed transactions to work off the copy. Nodes copied for reading provided repeatable read semantics while nodes copied for writing allowed writers to proceed regardless of simultaneous readers. When the transaction is about to commit, the modified nodes are compared with the versions in the main tree to avoid concurrent writes. If there are no conflicts, the modified nodes are finally merged with the main tree. With this option the degree of concurrency is increased. However, the conflicts are detected in commit time so a lot of costly processing time can be wasted. In the recent versions, JBoss Cache has replaced the pessimistic and optimistic approaches by a multi-version concurrency control mechanism similar to the one presented in Chapter 4. The idea is to allow readers continue reading shared state, while writers copy the shared state, increment a version id, and write that shared state back after verifying that the version is still valid (i.e., another concurrent writer has not changed this state first). As our approach, it detects conflicts early in concurrent transactional executions. However, JBoss Cache does not offer the advantages of the snapshot isolation level provided by our cache, and only guarantees the read committed and repeatable read isolation levels. Moreover, our validation approach to guarantee the consistency of the replicas in a cluster based on the timestamp of each transaction is more lightweight that the 2PC protocol used by JBoss Cache.

6.1.2.2.3. Terracotta

The approach of Terracotta [Ter] is different than the other cache mechanisms presented above. Terracotta installs a set of libraries on top of the JVM that provide clustering facilities at a lower level than the other caching solutions.

The architecture of a cluster of Terracotta is shown in Figure 6.1. The main elements are: (1) *Client Nodes* - A client node corresponds to a replica in the cluster. Client nodes are started in a standard JVM. When the client node is started, the Terracotta libraries are loaded. (2) *Terracotta Server Cluster* - It is composed of a set of Terracotta server replicas organized following a primary-backup approach. The primary Terracotta server acts as a highly-available second-level cache keeping synchronized shared accesses to memory. (3) *Storage* - The Terracotta Server Cluster uses disk storage as virtual heap storage. When the server can not store more objects paged-out from the client nodes (e.g. because the server's heap is filled), the server persist the objects onto disk. This also allows the primary to share the state with the backup server(s).

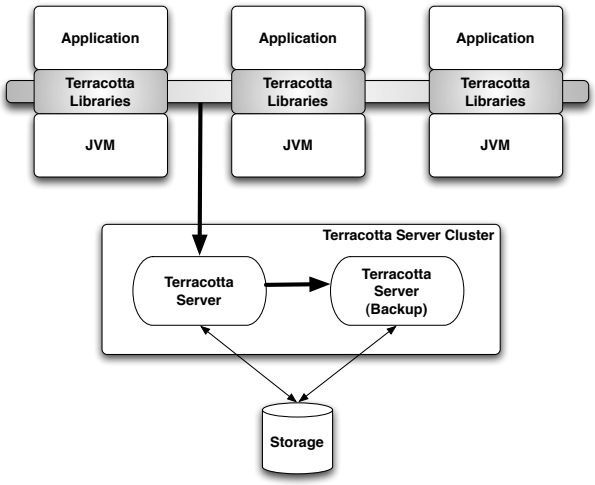


Figure 6.1: Terracotta architecture

The architecture shows that Terracotta is designed as a two-level cache. That is, the Terracotta server may persist or not the in-memory object data of the cluster to an on-disk representation. As the cluster uses a primary-backup approach, if the object data is persisted, the Terracotta server can survive failures. All the Terracotta server replicas must be configured to write their data to a shared filesystem that supports file locks. When the active server fails, one of the backups is promoted to be the new active server and the clients will automatically reconnect to the new active server. However, as the storage to persist cached data is shared among all the Terracotta servers, it is a potential point of failure for the cluster. The architecture does not allow to interpose either an object/relational mapper to retrieve/store the object data from/to a relational database.

Object sharing in Terracotta is done by means of the so-called "roots". An object that needs to be clustered is assigned to a root. Moreover, all the objects reachable from that object become also clustered objects. The changes to clustered objects can be made in all the client nodes (update-everywhere approach) and must be made within a transaction. When the

transaction is committed, all of the object data is sent to the Terracotta server. Terracotta uses cluster-wide locking semantics to guarantee concurrent execution of transactions. However, it uses a distributed locking mechanism based on the standard Java synchronization mechanism (which is based on mutual exclusion even for read operations) that compromises the scalability of the solution.

Integrating clustering facilities at JVM level avoid serialization code in clustered objects. Terracotta also allows fine-grained modifications in object fields to be transmitted to the replicas of the cluster. However, in order to prepare an application to be deployed in a cluster, the developer must know the internals of the application (class names, attributes...). In order to specify which data will be cached and replicated, the concrete packages, classes, attributes... must be specified in a configuration file. This can be a drawback because it does not allow to deploy in a cluster proprietary applications without having access to the source code.

6.1.2.2.4. Other Caches

Many other caches –either legacy or open-source– have arisen in the last few years: EHCache [Sou], GCache [Apaf], WhirlyCache [Jav], OSCache [Ope], ActiveSpace [Cod], GigaSpaces [Gig], NCache [Ala]... Some of the projects seem to be discontinued (GCache, WhirlyCache, OSCache, ActiveSpace...). These caches include features already described in the other caches so they are not detailed here.

6.1.2.3. Advanced Transaction Models

Regarding the support in the industry for advanced transaction models in multi-tier architectures, there have been several standardization efforts. The first one was proposed for CORBA, and was called CORBA Activity Service [Obja, HLR+01, HLR+03]. The service provides a general purpose event signaling mechanism that can be configured to enable activities –that may be seen as transactional units of work defined by applications– to coordinate each other in a manner prescribed by the model under consideration. In this way, advanced transaction models can be mapped onto specific implementations of the activity service framework.

The J(2)EE Activity Service [Sun06b] specification described and used in this Ph.D. Thesis (See chapters 2 and 3) is based on the work done for CORBA. The WebSphere application server has included the J2EE Activity Service in 5.x versions [IBM04] through the *ActivitySession* service. Apart from our implementation for the JBoss application server [Objc], WebSphere has been the only commercial application server that supports it.

6.2. Related Work on Service-Oriented Architectures

The next two sections describe the related work done in service-oriented architectures.

6.2.1. Scientific Work

The techniques used to provide high availability to web services exhibit some similarities with other middleware platforms. However, up to now, the related scientific work on this topic

is not frequent.

In [BvRV04], an architecture for web services availability has been presented. The proposed system does not rely exclusively on web service technology. Instead, it relies on TCP to provide group communication across nodes. They provide a (LAN) clustered intermediary message queue for reliable message exchange that is resilient to node failures but not to WAN connectivity failures. WS-Replication provides a more complete infrastructure that provides highly available web services and deals with replica failures.

The ad-hoc replication of a particular web service has been studied in [SLK04]. The paper presents an implementation of a replicated UDDI registry using TCP-based group communication that it is compared against the replicated UDDI specification [OASa]. There are two main differences between this work and ours. The first one is that it is a TCP-based solution. Thus, they do not rely exclusively on web service technology. However, unlike the previous approach, they deal with connectivity failures since they perform WAN replication as we do. The other difference is that the reported approach is particular for UDDI and not a generic framework for WAN replication of web services like the one provided by WS-Replication.

Ye and Shen describe in [YS05] one of the first attempts to provide a middleware for replicated web services. Their approach uses –as we do– active replication to achieve high availability. The middleware ensures that the clients' requests are sent to the replicas in the same order and that the code running in the replicas is deterministic (e.g. considering solutions for sources of non-determinism such as multithreaded web services). The internals of their architecture is similar to our model. Their middleware has a message handler element and a group communication manager implementing a total order protocol that are similar to our WS-Dispatcher and WS-Multicast elements. The performance tests included measure the overhead of their platform using a dummy web service simulating different operation times, but they do not include real scenarios nor WAN deployments and measures.

Thema [MIM+05] introduces another middleware for building fault tolerant web services. This middleware does not modify the semantics of web services standards, only requiring the use of WSDL and SOAP. The main contribution of this paper is the support of byzantine fault tolerance (BFT) [LSP82], allowing other failures that crash faults, such as software bugs or security violations. As we do, their replicated web services must guarantee the internal state consistency and support access from non-replicated clients (the clients must treat the replicated web service as a single entity) and access to non-replicated web services (the external web services must treat the replicated web service acting as a client as a single entity). They have tested the impact of providing BFT web services using the TPC-W benchmark [Tra05]. Instead of using HTTP as a transport, they compare a non-replicated version of TPC-W that uses SOAP communication over TCP, against a version that replicates a tier of the TPC-W application with BFT (SOAP communication over UDP). Moreover, this work does not provide the facilities offered by our framework in order to deploy easily the replicas of the web services in their corresponding nodes.

In [OFWG07], the authors present a similar approach to the work we presented in Chapter 5 for replication of stateful services using the Axis2 SOAP engine [Apad]. They also use a mixed approach between primary-backup and active replication. Client requests are delivered to a primary replica. Then, their replication framework multicast transparently the requests to the other backup replicas using a GCS (Spread). Requests are processed in each replica in order to achieve determinism and finally, only one response –the one produced at the primary– is filtered and returned to the client. The evaluation of the prototype is a little bit scarce

and it has been done only in a LAN environment. The authors claim that replication is not applied across organizational boundaries. However, we have shown in this Ph.D. Thesis that replication is also useful in inter-enterprise WAN environments to achieve high availability for critical services such as transactional engines.

The authors of [SS08] also aim to provide dependability and consistency for web services by means of a fault-tolerant architecture (WS-FTA). WS-FTA includes a set of components that implement the required features by the fault-tolerant architecture including replication, group communication, recovery... WS-FTA also claims that preserves the standards of the web service technology and allows legacy consumers to invoke replicated web services transparently. The architecture includes the ability to combine multiple groups of several replicas in order to deal with different types of failures. However, the evaluation of the prototype presented is very simple and has been done with a in a non-real scenario.

Two early papers that aimed to improve the dependability of web service compositions are [MTR02, TKM04]. Reliability of web services through transactions was proposed initially in [MTR02]. As web services are autonomous resources, different and potentially incompatible transaction models may be involved in the same service composition. The authors propose a middleware framework called WSTx for building reliable transactional compositions from web services with diverse transactional behavior. Through transactional attitudes, service providers and clients are able to describe their transactional semantics and requirements. As all the previous transactional approaches in other areas, this work handles data consistency. However, WSTx does not ensure other properties such as transparency (if a web service in the composition fails, there are no mechanisms to replace the failed web service), availability (no replication mechanism is supported) and scalability (the middleware receives all incoming requests being potentially a bottleneck). This work was continued by the authors in [TKM04] where their previous knowledge was adapted to the contexts of the new standard specifications for web services that arose in the meantime (BPEL, WS-Policy, WS-Coordination/WS-Transaction...).

Regarding the transactional support in SOAs, the theoretical studies on how to model the atomicity and isolation properties of transactions to improve the performance of applications [CR90] and the work on advanced transaction models [JK97, Elm92] and their applications (e.g. to workflows [AAE+96, WS97]) have crystallized in the current specifications such as BTP, WS-Coordination/WS-Transaction or WS-CAF. These specifications are described in the next section.

6.2.2. Industrial Work

The increasing interest for SOAs in the enterprises has created a lot of attention in the software industry. Many application server vendors –such as IBM, BEA and ORACLE– have adapted the features of their products to the service-oriented area. In [PvdH07] are discussed the benefits that application servers can offer to SOAs and also their limitations.

A lot of WS-* specifications have arisen in order to fulfill the requirements of service-oriented applications. W3C [W3Cb] and OASIS [OASc] are the organizations that control the development and adoption of the standards related to web services and SOAs. The main WS-* specifications related to the topics included in this thesis are described in the following paragraphs.

With regard to transactions in web services, the main specifications are the Business Transaction Protocol (BTP) [OAS02] and more recently, the WS-CAF [OAS05a] and WS-

Transaction and WS-Coordination [OAS05b].

The Business Transaction Protocol (BTP) [OAS02] was the first attempt to define a coordination protocol for web service-based transactional applications. It was submitted to the OASIS consortium to pass the standardization process. In BTP, each system that participates as a service consumer or service provider in a business transaction is composed of two elements (See Figure 6.2): an application element and a BTP element. The application elements exchange messages to accomplish the business function. When the bill payment service of a bank sends a message to the check writing service with details of the payee's name, address, and payment amount, the application elements of the two services are exchanging a message. The BTP elements of the two services also exchange messages that help compose, control, and coordinate a reliable outcome for the message sent between the application elements.

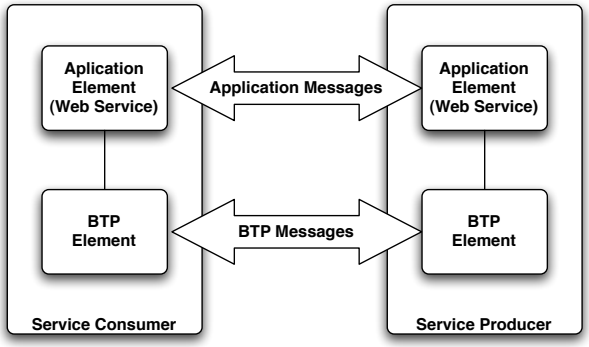


Figure 6.2: BTP elements

BTP uses a two-phase commit (2PC) coordination protocol to ensure the overall application achieves a consistent outcome. BTP permits the consistent outcome to be achieved by means of (1) *Atomic Business Transactions (Atoms)* - These are like traditional transactions, with a relaxed isolation property; and (2) *Cohesive Business Transactions (Cohesions)* - These are transactions where both isolation and atomicity properties are relaxed.

In the end, the BTP protocol was not supported by many of the principal software vendors and it was finally deprecated. When the BTP protocol become outdated, two new specifications related to transactions started the standardization process in the OASIS consortium: WS-CAF [OAS05a] and WS-Coordination/WS-Transaction [OAS05b]. Both specifications were too similar and presented overlapping concepts [NL04], so the OASIS consortium decided to merge the efforts of the two committees in charge –which included IBM, Microsoft, BEA, ORACLE, IONA and SUN– into a unique committee to continue the standardization process. Finally, the WS-Coordination/WS-Transaction specification was approved with the support of the principal software vendors.

The WS-CAF features have been introduced in Chapter 2. The main difference between the WS-CAF and WS-Transaction/WS-Coordination specifications is that the last one does not include the concept of "context" as an abstract first class entity. In the WS-CTX specification, WS-CAF defines the context as an abstract entity that can be specialized by top-level specifications in order to model security, transactions... () In WS-Transaction/WS-Coordination, a particular non-abstract context for transactions is defined ad-hoc in the WS-Coordination specification.

Several specifications that tackle different aspects of web service dependability have been

developed. For instance, WS-Reliable Messaging[13] [OAS04] deal with QoS when delivering SOAP messages. WS-Reliable Messaging is needed because SOAP over HTTP is not sufficient when an application-level messaging protocol must also guarantee some level of reliability and security. The protocol provided by the specification enriches SOAP with end-to-end delivery guarantees, including FIFO ordering and exactly-once, at most once, at least once semantics. The WS-Multicast web service of our WS-Replication framework allows to provide end-to-end delivery guarantees for SOAP messages based on the guarantees provided by the underlying group communication system, in this case JGroups.

Finally, just to point out that, to the best of our knowledge, none of the web service specifications have addressed the availability issue yet. Almost all software vendors have integrated into their enterprise suites support for service-oriented applications [NPT+07]. However, none of them currently include high available solutions for stateful web services. WS-Replication addresses this issue and provides a framework for replication of web services based exclusively on web service technology.

6.3. Other Technologies and Approaches

Software as a service (SaaS) is a new model of software deployment that is currently gaining a lot of attention. In this model, the applications are hosted by providers of hardware/software infrastructures (usually big companies) and offered as a service to the customers across the Internet. To implement this model, enterprise grids and compute clouds are used to provide the required QoS to the customers in terms of availability and scalability. Here, two important efforts of the industry in grid and cloud computing are described. They have come to light very recently (2007-2008): Amazon's Elastic Compute Cloud [Ama08a] and Google Application Engine [Goo]. They claim to offer easy-to-achieve availability and scalability for applications in their respective infrastructures. However is fairly unlikely that a single hosting company can create such kind of "elastic" infrastructures capable of fulfill the requirements of all kind of applications (e.g. to preserve data integrity, to deal with a massive amounts of users that access at all times and from different locations, to serve an increasing number of services...).

6.3.1. Amazon Elastic Compute Cloud

Elastic Compute Cloud (EC2)[14] [Ama08a] is a commercial web service for application providers that offers resizable computing capacity in a computer cloud owned by Amazon. Their objective is to provide an easy-to-manage scalable virtual computer infrastructure to deploy applications in which the application provider pays only for the resources that actually consumes, like instance-hours or data traffic. Moreover, in order to complement the computing facilities of EC2, Amazon also offers storage an data management options through the Amazon Simple Storage Service (Amazon S3) [Ama08b] and SimpleDB [Ama08c] respectively.

EC2 allows to manage the complete virtual computing environment through the use of web service interfaces. Through these interfaces, the application provider is able to create images

[13] Prior to WS-Reliable Messaging, OASIS produced a competing standard (WS-Reliability) that was supported by a coalition of vendors: Fujitsu, Hitachi, NEC, Oracle Corporation, Progress Software, and Sun Microsystems. Most of these vendors now also support the WS-Reliable Messaging specification.

[14] In May of 2008 EC2 is currently in Beta.

containing a ready-to-use custom application, add machines to the virtual environment, load them with a previously-created application image, manage access permissions...

With regard to reliability and scalability, EC2 offers the Amazon's highly reliable and powerful network infrastructure and data centers to run the virtual environments rented by the application providers. Failed instances can be replaced quickly and reliably commissioned in the virtual environment. Moreover EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios; it provides the ability to place instances in multiple locations geographically dispersed to protect applications from failures of a single location. In order to redirect request to the different locations, EC2 offers the so-called *elastic IP addresses*. These IP addresses allow the developers to mask instance or zone failures by programmatically remapping the public IP addresses to any other instance. In addition to instance replacement when failures occur, the application providers can increase or decrease capacity of their configurations when necessary within minutes.

However, with this approach, the ability to guarantee consistency in the data managed by applications in the cloud, the ability to scale out the application and the ability to recover applications from failures seem to rely on the skills of the application developers using the EC2 APIs and not in the features provided by the virtual environment itself. Thus, in contrast to the approach of this thesis EC2 does not offer a cluster infrastructure with the protocols to achieve state/data consistency, high availability or scalability for the applications running in in a distributed environment, but only the resources to achieve them.

6.3.2. Google Application Engine

With the introduction of Google Application Engine (GAE) at the beginning of 2008 [Goo], Google has gone one step further than Amazon. GAE allows the developers to implement and run web applications on Google infrastructure as Amazon does. However, GAE provides to developers a SDK[15] that simulates GAE and its environment in their local computers. When the application is ready, the developers may deploy it in the remote Google servers. Each application runs in a restricted "sandbox" environment. In this environment, the application can access the services and take advantage of the features that GAE provides. The main features provided by GAE are: dynamic web serving, with full support for common web technologies; persistent storage with queries, sorting and transactions; automatic load balancing and scaling[16]. Google claims that with this approach, the developers do not have to deal with server maintenance issues and allow their applications to stay always available and be scaled-out.

To ensure application availability, any GAE application runs on many web servers simultaneously. For the sake of simplicity and efficiency, applications can not spawn sub-processes or threads, as it happens in J(2)EE application servers. Client requests are managed by the environment and they can be redirected to any web server (GAE behaves as a load-balancing cluster). Thus, multiple requests from the same user may be handled by different web servers because no session data is associated to each client. Client requests that take long time to respond are automatically terminated to avoid overloading the web server. This means that long running activities are not allowed in GAE.

[15] The current language used in GAE is Python. In the near future it will be possible to use other languages.

[16] It has to be taken into account that, due to the internal policy of Google, there is not much information about the GAE internals.

The core service of GAE is its scalable transactional *datastore*. The datastore provides similar features as relational databases, but unlike them, it uses a distributed architecture to scale to very large data sets. A GAE application can optimize how data is distributed by describing relationships between data objects it must store, known as *entities* (similar to the entity beans of J(2)EE). Applications can put the entities in groups when they are created. The transaction manager of the datastore uses optimistic locking to deal with concurrent accesses of users to the same entities. A transaction is retried a fixed number of times if other transactions are trying to update the same entity simultaneously. Each transaction is limited to manipulate only entities that belong to the same group. The datastore stores together the entities of the same group to execute transactions efficiently. This fact, joint with the fact that the application state is only maintained in the entities (there are no sessions), suggest that application data is not replicated in the cluster, so it is not clear what happens with the entities stored in a node when it crashes.

GAE also provides a cache service called *Memcache*. Memcache implements a high performance in-memory key-value cache that is accessible transparently by multiple instances of the same application. It is useful for storing data that does not need persistence and require high-speed access. However, Memcache does not provide transactional features, so concurrent accesses to the same data can cause problems when they are not controlled specifically by the developer.

To conclude with, since GAE is such a controlled and limited environment, some drawbacks that are not present in clusters based on application servers arise: only incoming requests through HTTP and HTTPS protocols are accepted, the accesses to external computers are only allowed through the provided URL fetch and email services, it does not allow to write to the filesystem (all persistent data must be written into the datastore)...

CHAPTER 7

Conclusions

There is no evil that would last for 100 years nor body that would resist it.

– Anonymous (*Spanish proverb*)

Throughout chapters 3, 4 and 5, several replication and recovery protocols aiming to achieve the Ph.D. Thesis objectives outlined in Chapter 1 have been described and evaluated. This final chapter of the thesis summarizes the contents of these chapters. To conclude with, the chapter provides a vision of the work that can be done in the near future with regard to the topics and related areas discussed along the thesis.

7.1. Overall Summary

Nowadays, there are a lot of applications based in multi-tier and service-oriented architectures deployed at the core of any enterprise information system. The increasing requirements of some applications with regard to the reliability and performance of the infrastructures where they are running have motivated the work done in this Ph.D. Thesis. Examples of these kinds of applications the so-called extreme transaction processing (XTP) applications, which have to deal with a higher amount of transactions and have a wider scope (often multi-enterprise, national or world-wide) than traditional transactional applications. XTP applications have strong requirements on high availability and scalability and must be deployed in high performance clusters or grids. Moreover, the infrastructures offering software as a service (SaaS) –such as Google Application Engine or Amazon Elastic Compute Cloud– also require to guarantee the high availability and performance of the applications deployed by their clients.

The overall objective of the Ph.D. Thesis has been to design and evaluate a set of protocols to provide *consistency*, *high availability* and *scalability* to applications deployed on replicated multi-tier and service-oriented architectures. The thesis comprises several contributions to the areas of component and service replication and transaction processing that allow to achieve these non-functional properties. The contributions presented have been implemented and integrated with J(2)EE and web service technology based on SOAP, the most used middleware platforms for implementing multi-tier and service-oriented architectures in current enterprise information systems.

In order to provide high availability to multi-tier architectures, the state of the application components has been replicated transparently to the clients to several replicas in a cluster. Several replication protocols that provide high availability have been presented. These protocols take into account the (complex) interaction patterns from the clients, guaranteeing the exactly-once execution of the requests. The protocols are also transaction-aware, so they also guarantee the state consistency of the application components in all the replicas.

Another transaction-aware protocol that provides not only high availability, but also scalability to multi-tier architectures has been presented. The protocol guarantees one-copy snapshot isolation in a high performance cluster. Online recovery is a complementary issue to provide high availability and scalability, so the online recovery of the state of the replicas has also been studied in multi-tier architectures.

Finally, high availability in service-oriented architectures (SOAs) have been also studied in depth in this Ph.D Thesis. A framework that eases the replication of critical web services in SOAs has been proposed and successfully evaluated. The replication process preserves the web services autonomy and is transparent to the clients.

The following sections summarize the tasks done in the Ph.D. Thesis with regard to each architecture.

7.1.1. Multi-tier Architectures

Chapter 3 has presented a novel replication support to provide high availability to multi-tier J(2)EE applications. Current solutions for high availability provided by application servers replicate the stateful components of the business tier and share a common instance of the database. However, these solutions do not provide high availability for the whole architecture because, if the DBMS fails, the applications cannot progress. With the vertical replication model presented in the thesis, both tiers, the business tier (application server state) and the data tier (database), are replicated together in a single step. This eliminates the possibility for the database of being a single point of failure.

Three different replication protocols suitable for different interaction patterns have been described and evaluated: (1) a basic protocol for simple transactional invocations (1 request/1 transaction); (2) an evolution of the first protocol supporting multiple client invocations on the same transaction (N requests/1 transaction) and finally, (3) a protocol supporting long running activities with complex interaction patterns (1 request/N transactions and N requests/M transactions). The protocols allow us to configure a high availability cluster based on a primary-backup replication approach. Moreover, they extend highly availability to transactions and long running activities, what enables state consistency in all the replicas of the cluster.

The main innovations of this approach are: (1) the replication process is transaction and activity aware; (2) transactional contexts can deal with complex client interactions and finally,

(3) failure masking is transparent to both, the applications running on top of the replication framework and the clients of that applications (that is, when a failure occurs in the primary replica, the processing continues in the new primary at the point where the previous primary failed).

The ADAPT replication framework has been used on top of the JBoss open source application server in order to implement the replication protocols. Moreover, a prototype of the activity service specification has been implemented. On top of the activity service, the open nested transaction (ONT) model has been implemented in order to evaluate the replication protocol with support for long running activities[1].

The ECPerf benchmark has been used for evaluating the first two protocols. The ECPerf's client application generates the simulated workload for the system under test. In order to evaluate the protocol for long running activities, a specific ECPerf-based application and an application for injecting workload have been created. The replication protocols have exhibited good performance compared with the non-replicated ones. The overhead introduced by the framework and the replication protocols worth the high availability features that they provide. The results of this work have been also published in [PPJV06].

In Chapter 4 the replication infrastructure for multi-tier architectures has been taken one step further. Instead of using a primary-backup approach, an update-everywhere schema allows all the replicas to receive update requests from the clients. The solution proposed uses a replicated cache with snapshot isolation for application server components and the vertical replication approach to synchronize the changes with the database. The protocol is also transaction-aware, so it guarantees data consistency in the cluster and exploits snapshot isolation to avoid the contention due to read/write conflicts in concurrent transactions. This allows us to configure a high performance cluster with one-copy snapshot isolation guarantees and providing high availability and scalability to multi-tier applications.

The implementation of the prototype has been done using JOnAS, an open source J(2)EE application server. The multi-version cache has been integrated in its EJB container and the replication protocol has been deployed as a new service of the application server.

The SPECjAppServer industrial benchmark for J(2)EE application server has been used to perform a thorough evaluation of the resulting system. SPECjAppServer is an up-to-date version of the ECPerf benchmark. The results obtained during the evaluation have demonstrated the good behaviour that the protocol exhibits with regard to scalability. This work has been published in [PPJK07].

This replication protocol is complemented by a recovery protocol that is able to recover failed replicas and to incorporate totally new replicas to scale out a high performance cluster. During the recovery phase, the cluster does not stop processing new client requests, what is called online recovery. Moreover, all the online replicas of the cluster participate in the recovery process of the new replica and continue serving client requests at the same time. At the end of the recovery process, the final state of the replica that is being incorporated to the cluster is consistent with the other replicas at both tiers, application server and database.

The evaluation of the recovery prototype has also been done using the SPECjAppServer benchmark. The results of the experiments done with the different sizes of the state required to recover a replica have shown the expected outcomes.

[1]Both implementations, the J2EE Activity Service and the ONT model, are available at ObjectWeb as part of the JASS open source project: http://forge.objectweb.org/projects/jass/

7.1.2. Service-Oriented Architectures

With regard to the work done in service-oriented architectures (SOAs), Chapter 5 has described the WS-Replication framework. This framework allows us to provide high availability to critical web services by means of replication. The framework takes non-replicated web services as input and automatically generates and deploys replicas of the web service in a set of nodes. The nodes containing the replicas of a service can be located in the same LAN or in different LANs connected through WANs. Moreover, the WS-Replication framework provides transparent replication. That is, the clients of a replicated web service do not know that they are interacting with a replicated service. In order to respect the web service autonomy, WS-Replication includes a core web service called WS-Multicast that provides group communication facilities over the SOAP protocol. WS-Multicast that can also be used outside the framework as a standalone application component for providing reliable multicast to SOAs.

In order to evaluate the WS-Replication framework, the Web Services Composite Application Framework (WS-CAF) has been implemented and replicated using the framework functionality. WS-CAF is a framework that provides transactional support to SOAs and represents an example of a critical web service requiring high availability in a service-oriented infrastructure. The supply chain management application described in the WS-I Basic Profile 1.0 has been implemented as example application using the replicated WS-CAF. This application has been adapted to include the WS-CAF's long running activities (LRAs) transaction model in its business logic[2].

The web services infrastructure required in the evaluation has been deployed in JBoss application server instances with the Axis web services engine. Several experiments in both, LAN and WAN environments, have been done to evaluate the WS-Replication framework. SOAP communication has also been contrasted with TCP communication. The different evaluations have shown that the overhead of the SOAP-based replication framework is more than acceptable taking into account the benefits that the framework adds to SOAs. This work on web services replication has been published in [SPPJ06].

7.2. Future Work

The work done in this Ph.D. Thesis constitutes a first step in what can be achieved with regard to high available and scalable middleware for multi-tier and service-oriented architectures. The following paragraphs outline only a few ideas about what can be done in the near future.

First of all, it is worth pointing out again that the protocols that constitute the core of this Ph.D Thesis can be adapted and implemented in other middleware platforms and technologies available today, such as in lightweight containers, .NET, CORBA... Moreover, in the near future, new and potentially disruptive platforms supporting new programming models (e.g. multi-core, grid-based...) will appear as alternatives to CORBA, .NET and J(2)EE platforms. These platforms will also have strong requirements on availability, scalability and consistency. It is quite possible that the protocols and ideas described here could also be adapted to these new platforms.

[2]The LRA implementation of WS-CAF can be found at ObjectWeb as part of the JASS open source project: http://forge.objectweb.org/projects/jass/

The protocol presented in Chapter 4 could be modified to reduce the number of database replicas. The objective of this approach is to minimize the cost in licenses when using expensive commercial database management systems (e.g. Oracle, SQL server...). As in the current protocol, the idea is to keep consistent the caches in all the replicas at the application server level. However, instead of using a vertical replication approach in each node, certain application server replicas behave as coordinators to drive the persistence process of the cached components in the database instances. This way, despite not using a vertical replication schema in all the cluster replicas, both, high availability and scalability, are still preserved in the multi-tier architecture.

With regard to replication in service-oriented middleware, scalability for web services is still an open issue. Scalability is the key to not to turn the critical services of a service-oriented infrastructure into bottlenecks for the applications.

Finally, partial replication techniques for the state managed in the whole cluster can also be used to define new replication protocols. Partial replication exploits data locality to improve the performance of the queries on high amounts of data. As the individual replicas do not spend processing time applying the updates from all the replicas belonging to the cluster, the saved CPU time can be used to process additional local requests.

I have made this [letter] longer, because I have not had the time to make it shorter.

– Blaise Pascal

References

[AAE+96] Gustavo Alonso, Divyakant Agrawal, Amr El Abbadi, Mohan Kamath, Roger Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 574–581. IEEE Computer Society Press, 1996.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architecture and Applications*. Springer Verlag, 2004.

[ADA] ADAPT Project IST-37126. *ADAPT: Middleware Technologies for Adaptive and Composable Distributed Components*. http://adapt.ls.fi.upm.es/adapt.htm.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A Communication Subsystem for High Availability. In *Proc. of the Int. Symp. on Fault Tolerant Computer Systems (FTCS)*, pages 76–84, 1992.

[ADR00] Yair Amir, Claudiu Danilov, and Jonathan Robert Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, pages 327–336. IEEE Computer Society Press, 2000.

[Ala] Alachisoft. *NCache*. http://www.alachisoft.com/ncache/.

[Ama] Amazon Inc. *Amazon Web Services*. http://aws.amazon.com/.

[Ama08a] Amazon Inc. *Elastic Compute Cloud*, 2008. http://aws.amazon.com/ec2/.

[Ama08b] Amazon Inc. *Simple Storage Service*, 2008. http://aws.amazon.com/s3/.

[Ama08c] Amazon Inc. *SimpleDB*, 2008. http://aws.amazon.com/simpledb/.

[AMD+06] José Enrique Armendáriz-Iñigo, Francesc D. Muñoz-Escoí, Hendrik Decker, J. R. Juárez-Rodríguez, and José Ramón González de Mendívil. A Protocol for Reconciling Recovery and High-Availability in Replicated Databases. In *Proc. of the Int. Symp. on Computer and Information Sciences (ISCIS)*, volume 4263 of *Lecture Notes in Computer Science*, pages 634–644. Springer Verlag, 2006.

[AÖ06] M. Hossein Sheikh Attar and M. Tamer Özsu. Alternative Architectures and Protocols for Providing Strong Consistency in Dynamic Web Applications. *World Wide Web Journal*, 9(3):215–251, 2006.

[Apaa] Apache Foundation. *Apache CXF: An Open Source Service Framework*. http://incubator.apache.org/cxf/.

[Apab] Apache Foundation. *Apache HTTP Server*. http://httpd.apache.org/.

[Apac] Apache Foundation. *Axis SOAP Engine*. http://ws.apache.org/axis/.

[Apad] Apache Foundation. *Axis2 SOAP Engine*. http://ws.apache.org/axis2/.

[Apae] Apache Foundation. *Clustering in Geronimo Application Server*. http://cwiki.apache.org/GMOxDEV/clustering.html.

[Apaf] Apache Foundation. *GCache*. http://cwiki.apache.org/GMOxDEV/caching.html and http://cwiki.apache.org/GMOxDEV/geronimo-clustering-with-gcache.html.

[Apag] Apache Foundation. *Geronimo Application Server*. http://geronimo.apache.org/.

[Apah] Apache Foundation. *HiveMind*. http://hivemind.apache.org/.

[BB99] Mark Baker and Rajkumar Buyya. Cluster Computing at a Glance. In R. Buyya, editor, *High Performance Cluster Computing: Architecture and Systems, Volume 1*, pages 3–47. Prentice Hall, 1999.

[BBD97] Özalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 46(6):642–658, 1997.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 1–10. ACM Press, 1995.

[BBM+04] Özalp Babaoglu, Alberto Bartoli, Vance Maverick, Simon Patarin, Jaksa Vuckovic, and Huaigu Wu. A Framework for Prototyping J2EE Replication Algorithms.

In *Proc. of the OTM Confederated Int. Conf. On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1413–1426. Springer Verlag, 2004.

[BEA] BEA Systems Inc. *WebLogic Application Server*. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic&WT.ac=topnav_products_weblogic.

[BEA07] BEA Systems Inc. *Using WebLogic Server Clusters*, 2007. http://e-docs.bea.com/wls/docs100/pdf/cluster.pdf.

[Ber] Berkeley/Stanford. *Recovery-Oriented Computing*. http://roc.cs.berkeley.edu/roc_overview.html.

[BFG+06] Philip A. Bernstein, Alan Fekete, Hongfei Guo, Raghu Ramakrishnan, and Pradeep Tamma. Relaxed-currency Serializability for Middle-tier Caching and Replication. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 599–610. ACM Press, 2006.

[BG83] Philip A. Bernstein and Nathan Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1987.

[Bir97] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., Greenwich, CT, USA, 1997.

[BJK+05] A. Bartoli, R. Jimenez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman. The Adapt Framework for Adaptable and Composable Web Services. *IEEE Distributed Systems Online*, September 2005.

[BLBA00] Roger S. Barga, David B. Lomet, Thomas Baby, and Sanjay Agrawal. Persistent Client-Server Database Sessions. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, volume 1777 of *Lecture Notes in Computer Science*, pages 462–477. Springer Verlag, 2000.

[BLSW04] Roger S. Barga, David B. Lomet, German Shegalov, and Gerhard Weikum. Recovery Guarantees for Internet Applications. *ACM Transactions on Internet Technology*, 4(3):289–328, 2004.

[BLW02] Roger S. Barga, David B. Lomet, and Gerhard Weikum. Recovery Guarantees for General Multi-Tier Applications. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 543–554. IEEE Computer Society Press, 2002.

[BM03] Roberto Baldoni and Carlo Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software: Practice and Experience*, 33(8):767–797, 2003.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal Primary-Backup Protocols. In *Proc. of the Int. Workshop on Distributed Algorithms (WDAG)*, volume 647 of *Lecture Notes in Computer Science*, pages 362–378. Springer Verlag, 1992.

[BN96] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann Publishers Inc., 1996.

[BP01] A. B. Brown and D. A. Patterson. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). In *High Performance Transaction Processing Symposium*, October 2001.

[BvRV04] Kenneth P. Birman, Robbert van Renesse, and Werner Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pages 17–26. IEEE Computer Society Press, 2004.

[BVZ09] Luiz E. Buzato, Gustavo M. D. Vieira, and Willy Zwaenepoel. Dynamic Content Web Applications: Crash, Failover, and Recovery Analysis. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE Computer Society Press, 2009. To appear.

[Can07] Michele Cantara. Report Highlight for User Survey Analysis: SOA, Web Services and Web 2.0 User Adoption Trends and Recommendations for Software Vendors, North America and Europe, 2005-2006. Technical report, Gartner Group, January 2007.

[CDK01] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CKV01] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):427–469, 2001.

[CL00] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 273–288. ACM Press, 2000.

[Cod] Codehaus. *ActiveSpace*. http://activespace.codehaus.org/.

[Cor] Cornell University. *The Isis Project*. http://www.cs.cornell.edu/Info/Projects/ISIS/.

[CR90] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 194–203. ACM Press, 1990.

[DSS98] Xavier Défago, André Schiper, and Nicole Sergent. Semi-Passive Replication. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 43–50, 1998.

[EJ] EJ Technologies. *JProfiler*. http://www.ej-technologies.com/products/jprofiler/overview.html.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers Inc., 1992.

[EMS95] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 296–306, 1995.

[FG01] Svend Frolund and Rachid Guerraoui. Implementing E-Transactions with Asynchronous Replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, 2001.

[FG02] Svend Frolund and Rachid Guerraoui. e-Transactions: End-to-End Reliability for Three-Tier Architectures. *IEEE Transactions on Software Engineering*, 28(4):378–395, 2002.

[FGS98] Pascal Felber, Rachid Guerraoui, and André Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[FN02] Pascal Felber and Priya Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *Proc. of the OTM Confederated Int. Conf. On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 737–754. Springer Verlag, 2002.

[Fow04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection Pattern*. ThoughWorks, Jan 2004. http://www.martinfowler.com/articles/injection.html.

[FP98] Jean-Charles Fabre and Tanguy Pérennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[FvR95] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1537, CS Dep., Cornell Univ., 1995.

[GHM+99] Rachid Guerraoui, Michel Hurfin, Achour Mostéfaoui, Riucarlos Oliveira, Michel Raynal, and André Schiper. Consensus in Asynchronous Distributed Systems: A Concise Guided Tour. In *Advances in Distributed Systems*, pages 33–47, 1999.

[GHOS96] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha. The Dangers of Replication and a Solution. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 173–182. ACM Press, 1996.

[Gig] GigaSpaces. *GigaSpaces*. http://www.gigaspaces.com/.

[GLPT76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[Goo] Google Inc. *Google Application Engine (GAE)*. http://code.google.com/appengine/.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1993.

[Gra81] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 144–154. IEEE Computer Society Press, 1981.

[GS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 249–259. ACM Press, 1987.

[Hay98] M. Hayden. The ensemble system. Technical Report TR-98-1662, Department of Computer Science. Cornell University, January 1998.

[HLR+01] Iain Houston, Mark C. Little, Ian Robinson, Santosh K. Shrivastava, and Stuart M. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. In *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, volume 2218 of *Lecture Notes in Computer Science*, pages 197–215. Springer Verlag, 2001.

[HLR+03] Iain Houston, Mark C. Little, Ian Robinson, Santosh K. Shrivastava, and Stuart M. Wheater. The CORBA Activity Service Framework for supporting extended transactions. *Software: Practice and Experience*, 33(4):351–373, 2003.

[HR83] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computer Surveys*, 15(4):287–317, 1983.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems (2nd Edition*, pages 97–145. ACM Press, 1993.

[IBM] IBM. *WebSphere Application Server*. http://www-306.ibm.com/software/webservers/appserv/was/.

[IBM04] IBM. *IBM WebSphere Application Server V5.1 System Management and Configuration*, 2004. http://www.redbooks.ibm.com/redbooks/pdfs/sg246195.pdf.

[IBM05] IBM. *WebSphere Application Server Network Deployment V6: High Availability*, 2005.
http://www.redbooks.ibm.com/redbooks/pdfs/sg246688.pdf.

[IBM06] IBM. *WebSphere Application Server V6.1: Planning and Design*, 2006.
http://www.redbooks.ibm.com/redbooks/pdfs/sg247305.pdf.

[IBM08] IBM. *WebSphere eXtreme Scale (WXS)*, 2008.
http://www-306.ibm.com/software/webservers/appserv/extend/.

[Int] Interface21. *Spring*.
http://www.springframework.org/.

[Iro] Ironflare AB. *Orion Application Server*.
http://www.orionserver.com/.

[Jav] Java.net. *Whirlycache*.
https://whirlycache.dev.java.net/.

[JGr] B. Ban. *JGroups*.
http://www.jgroups.org.

[Jim09] Ricardo Jimenez-Peris. Online Recovery in Parallel Database Systems. In M. Tamer Özsu and Ling Liu, editors, *Enclyclopedia of Database Systems*. Springer Verlag, 2009. To appear.

[JK97] Sushil Jajodia and Larry Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Press, 1997.

[JPA02] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Gustavo Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 150–159. IEEE Computer Society Press, 2002.

[KA00a] Bettina Kemme and Gustavo Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[KA00b] Bettina Kemme and Gustavo Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 134–143. Morgan Kaufmann Publishers Inc., 2000.

[KBB01] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, pages 117–130. IEEE Computer Society Press, 2001.

[KJPS05] Bettina Kemme, Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Jorge Salas. Exactly Once Interaction in a Multi-tier Architecture. In *Proceedings of VLDB Workshop on Design, Implementation and Deployment of Database Replication*, 2005.

[KMSL03] Achmad I. Kistijantoro, Graham Morgan, Santosh K. Shrivastava, and Mark C. Little. Component Replication in Distributed Systems: A Case Study Using Enterprise Java Beans. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 89–98. IEEE Computer Society Press, 2003.

[Lam98] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam06] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LBG04] Sacha Labourey, Bill Burke, and JBoss Group. *JBoss AS Clustering (3.x)*. JBoss Inc., 2004.
http://docs.jboss.org/jbossas/clustering/JBossClustering7.pdf.

[Lig] Codehaus. *PicoContainer*.
http://www.picocontainer.org/.

[Lit04] Mark Little. Models for Web Services Transactions. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, page 872. ACM Press, 2004.

[LK08] WeiBin Liang and Bettina Kemme. Online Recovery in Cluster Databases. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 121–132. ACM Press, 2008.

[LKPJ05] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *Proc. of the Int. Conf. on Management of Data (ACM SIGMOD)*, pages 419–430. ACM Press, 2005.

[LM06] Edmond Lau and Samuel Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 703–714. ACM Press, 2006.

[LR03] Avraham Leff and James T. Rayfield. Improving Application Throughput With Enterprise JavaBeans Caching. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 244–251. IEEE Computer Society Press, 2003.

[LR04] Avraham Leff and James T. Rayfield. Alternative Edge-Server Architectures for Enterprise JavaBeans Applications. In Hans-Arno Jacobsen, editor, *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 195–211. Springer Verlag, 2004.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[MHL+92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[Mica] Microsoft Corp. *Microsoft .NET Framework*. http://www.microsoft.com/net/.

[Micb] Microsoft Corp. *Microsoft Windows Server*. http://www.microsoft.com/windowsserver2003/default.mspx.

[Micc] Microsoft Corp. *Open Database Connectivity (ODBC)*. http://support.microsoft.com/kb/110093.

[Micd] Microsoft Corp. *Phoenix: Making Applications Robust*. http://research.microsoft.com/db/phoenix/.

[Mic03] Microsoft Corp. *Server Clusters: Frequently Asked Questions for Windows 2000 and Windows Server 2003*, 2003. http://www.microsoft.com/technet/prodtechnol/windowsserver2003/technologies/clustering/sercsfaq.mspx.

[MIM+05] Michael G. Merideth, Arun Iyengar, Thomas A. Mikalsen, Stefan Tai, Isabelle Rouvellou, and Priya Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 131–142. IEEE Computer Society Press, 2005.

[MMA+96] Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, 1996.

[MMN+99a] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, L.A. Tewksbury, and V. Kalogeraki. The Eternal System: An Architecture for Enterprise Applications. In *Enterprise Distributed Object Computing Conference (EDOC)*, pages 214–222, 1999.

[MMN99b] Louise E. Moser, P. M. Melliar-Smith, and Priya Narasimhan. A Fault Tolerance Framework for CORBA. In *Proc. of the Int. Symp. on Fault Tolerant Computer Systems (FTCS)*, pages 150–157, 1999.

[MMSW07] Maged M. Michael, José E. Moreira, Doron Shiloach, and Robert W. Wisniewski. Scale-up x Scale-out: A Case Study using Nutch/Lucene. In *Proc. of the Int. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8. IEEE Computer Society Press, 2007.

[Mos81] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1981. Also, Technical Report MIT/LCS/TR-260.

[MSEL99] Graham Morgan, Santosh K. Shrivastava, Paul D. Ezhilchelvan, and Mark C. Little. Design and Implemantation of a CORBA Fault-tolerant Object Group Service. In *Proc. of the Int. Conf. on Distributed Applications and Interoperable Systems*, volume 143 of *IFIP Conference Proceedings*, pages 361–374. Kluwer Academic Press, 1999.

[MTR02] Thomas Mikalsen, Stefan Tai, and Isabelle Rouvellou. Transactional Attitudes: Reliable Composition of Autonomous Web Services. In *Workshop on Dependable Middleware-based Systems (WDMS 2002), part of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society Press, 2002.

[Nar99] Priya Narasimhan. *Transparent Fault tollerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 1999.

[Nar07] Priya Narasimhan. Fault-Tolerant CORBA: From Specification to Reality. *IEEE Computer*, 40(1):110–112, 2007.

[New02] Eric Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[NL04] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[NMM01] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, pages 261–270. IEEE Computer Society Press, 2001.

[NMM02a] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Eternal - A Component-Based framework for Transparent Fault-Tolerant CORBA. *Software: Practice and Experience*, 32(8):771–788, 2002.

[NMM02b] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Lessons Learned in Building a Fault-Tolerant CORBA System. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, pages 39–44. IEEE Computer Society Press, 2002.

[NPT+07] Yefim V. Natis, Massimo Pezzini, Jess Thompson, Kimihiko Iijima, Michael Barnes, Daryl C. Plummer, and Simon Hayward. Magic quadrant for application infrastructure for new service-oriented business application projects, 2q07. Technical report, Gartner Group, May 2007.

[OASa] OASIS. *UDDI Version 2.03 Replication Specification*. http://uddi.org/pubs/Replication_v2.htm.

[OASb] OASIS. *Universal Description, Discovery and Integration(UDDI)*. http://uddi.org.

[OASc] OASIS Consortium. *OASIS*. http://www.oasis-open.org.

[OAS02] OASIS. *Business Transaction Protocol (BTP)*, 2002. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=business-transaction.

[OAS04] OASIS. *Web Services Reliable Messaging (WS-RM)*, 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.

[OAS05a] OASIS. *Web Services Composite Application Framework (WS-CAF)*, 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-caf.

[OAS05b] OASIS. *Web Services Transaction (WS-TX)*, 2005. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx.

[Obja] Object Management Group (OMG). *Additional Structuring Mechanisms for the OTS Specification 1.0*. http://www.omg.org/technology/documents/formal/add_struct.htm.

[Objb] Object Management Group (OMG). *CORBA/IIOP Specification*. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[Objc] Object Management Group (OMG). *Fault Tolerant CORBA*. http://www.omg.org/technology/documents/formal/corba_iiop.htm.

[Objd] Objectweb Consortium. *Clustering in JOnAS Application Server*. http://jonas.objectweb.org/JOnAS_4_7/doc/Cmi.html.

[Obje] Objectweb Consortium. *Java Advanced Transaction Support (JASS)*. http://jass.objectweb.org.

[Objf] Objectweb Consortium. *JOnAS Application Server*. http://jonas.objectweb.org.

[Objg] Objectweb Consortium. *RUBiS Benchmark*. http://rubis.objectweb.org/.

[OFWG07] Johannes Osrael, Lorenz Froihofer, Martin Weghofer, and Karl M. Göschka. Axis2-based Replication Middleware for Web Services. In *Proc. of the Int. Conf. on Web Services (ICWS)*, pages 591–598. IEEE Computer Society Press, 2007.

[OGP03] David L. Oppenheimer, Archana Ganapathi, and David A. Patterson. Why Do Internet Services Fail, and What Can Be Done About It? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[Ope] Opensymphony. *OSCache*. http://www.opensymphony.com/oscache/.

[Oraa] Oracle Corp. *Oracle Application Server*. http://www.oracle.com/appserver/index.html.

[Orab] Oracle Corp. - Tangosol. *Tangosol Coherence*. http://www.tangosol.com/coherence-overview.jsp.

[Ora06] Oracle Corp. *Oracle Application Server 10g Release 3 High Availability*, 2006. http://www.oracle.com/technology/products/ias/hi_av/OracleApplicationServer10gR3HA-WP.pdf.

[PBL91] D. Powell, I. Bey, and J. Leuridan, editors. *Delta Four: A Generic Architecture for Dependable Distributed Computing*. Springer Verlag, Secaucus, NJ, USA, 1991.

[PGNS07] Massimo Pezzini, Milind Govekar, Yefim V. Natis, and Donna Scott. Extreme transaction processing: Technologies to watch. Technical report, Gartner Group, February 2007.

[PN07] Massimo Pezzini and Yefim V. Natis. Trends in platform middleware: Disruption is in sight. Technical report, Gartner Group, September 2007.

[Posa] PostgreSQL Global Development Group. *Point In Time Recovery in PostgreSQL*. http://developer.postgresql.org/pgdocs/postgres/backup-online.html.

[Posb] PostgreSQL Global Development Group. *PostgreSQL DBMS*. http://www.postgresql.org/.

[PPJK07] Francisco Perez-Sorrosal, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Bettina Kemme. Consistent and Scalable Cache Replication for Multi-tier J2EE Applications. In *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, volume 4834 of *Lecture Notes in Computer Science*, pages 328–347. Springer Verlag, 2007.

[PPJV06] Francisco Perez-Sorrosal, Marta Patiño-Martínez, Ricardo Jiménez-Peris, and Jaksa Vuckovic. Highly Available Long Running Transactions and Activities for J2EE Applications. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, page 2. IEEE Computer Society Press, 2006.

[PvdH07] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *Very Large DataBases Journal*, 16(3):389–415, 2007.

[Reda] Red Hat. *JBoss Application Server*. http://www.jboss.com.

[Redb] Red Hat. *JBoss Cache*. http://labs.jboss.com/jbosscache/.

[Red06] Red Hat. *The JBoss 4 Application Server Clustering Guide*, 2006. http://docs.jboss.com/jbossas/guides/clusteringguide/r2/en/pdf/jboss4-clustering.pdf.

[RQC05] Paolo Romano, Francesco Quaglia, and Bruno Ciciani. A Lightweight and Scalable e-Transaction Protocol for Three-Tier Systems with Centralized Back-End Database. *IEEE Transactions on Knowledge and Data Engineering*, 17(11):1578–1583, 2005.

[RRQC08] Paolo Romano, Diego Rughetti, Francesco Quaglia, and Bruno Ciciani. APART: Low Cost Active Replication for Multi-tier Data Acquisition Systems. In *Proc. of the Int. Symp. on Networking Computing and Applications (NCA)*, pages 1–8. IEEE Computer Society, 2008.

[Sch84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computer Surveys*, 22(4):299–319, 1990.

[Sch93] Michael D. Schroeder. State-of-the-Art Distributed System: Computing with BOB. In Sape Mullender, editor, *Distributed Systems (2nd Edition)*, pages 1–16. ACM Press, 1993.

[SLK04] Chenliang Sun, Yi Lin, and Bettina Kemme. Comparison of UDDI Registry Replication Strategies. In *Proc. of the Int. Conf. on Web Services (ICWS)*, pages 218–225. IEEE Computer Society Press, 2004.

[Sou] Sourceforge. *EHCache*. http://ehcache.sourceforge.net/.

[SPH06] Sylvain Sicard, Noel De Palma, and Daniel Hagimont. J2EE Server Scalability Through EJB Replication. In Hisham Haddad, editor, *Proc. of the ACM Symp. on Applied Computing (SAC)*, pages 778–785. ACM Press, 2006.

[SPPJ06] Jorge Salas, Francisco Perez-Sorrosal, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. WS-Replication: a Framework for Highly Available Web Services. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, pages 357–366. ACM Press, 2006.

[Spr] Spread Concepts LLC. *The Spread Toolkit*. http://www.spread.org/.

[SS08] Jeferson L. R. Souza and Frank Siqueira. Providing Dependability for Web Services. In *Proc. of the ACM Symp. on Applied Computing (SAC)*, pages 2207–2211. ACM Press, 2008.

[Sta04] Standard Performance Evaluation Corporation. *SPECjAppServer Benchmark*, 2004. http://www.spec.org/jAppServer/.

[Suna] Sun Microsystems. *GlassFish Application Server*. https://glassfish.dev.java.net/.

[Sunb] Sun Microsystems. *Java Programming Language*. http://java.sun.com/.

[Sunc] Sun Microsystems. *Java Transaction Service (JTS)*. http://java.sun.com/products/jts/.

[Sund] Sun Microsystems. *JDBC Technology*. http://java.sun.com/javase/technologies/database/index.jsp.

[Sune] Sun Microsystems. JCP Program. *JSR-107: JCACHE - Java Temporary Caching API*. http://jcp.org/en/jsr/detail?id=107.

[Sun99] Sun Microsystems. *Java Transaction API Specification (JTA) v1.0*, April 1999. http://java.sun.com/products/jta/.

[Sun03a] Sun Microsystems. *ECperf Benchmark Specification v1.1 (Final Release)*, 2003. http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html.

[Sun03b] Sun Microsystems. *J2EE specification v1.4*, 2003.

[Sun06a] Sun Microsystems. *JEE specification v5*, 2006. http://java.sun.com/javaee/.

[Sun06b] Sun Microsystems. JCP Program. *JSR-95: J2EE Activity Service for Extended Transactions*, 2006. http://jcp.org/en/jsr/detail?id=95.

[Ter] Terracotta Inc. *Terracotta*. http://www.terracotta.org/.

[TKM04] Stefan Tai, Rania Khalaf, and Thomas A. Mikalsen. Composition of Coordinated Web Services. In *Proc. of the ACM/IFIP/USENIX International Middleware Conference*, volume 3231 of *Lecture Notes in Computer Science*, pages 294–310. Springer Verlag, 2004.

[Tra] Transaction Processing Performance Council. *Transaction Processing Performance Council*. http://tpc.org/.

[Tra05] Transaction Processing Performance Council. *TPC-W Benchmark*, 2005. http://www.tpc.org/tpcw/default.asp.

[VBH+91] P. Verissimo, P. Barret, A. Hilborne, L. Rodrigues, and D. Seaton. The Extra Performance Architecture (XPA). In D. Powell, editor, *Delta Four: A Generic Architecture for Dependable Distributed Computing*, pages 211–266. 1991.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, 1996.

[W3Ca] W3C. *Simple Object Access Protocol (SOAP) 1.1*. http://www.w3.org/TR/soap/.

[W3Cb] World Wide Web Consortium. *W3C*. http://www.w3.org.

[W3C01] W3C. *Web Services Description Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl.

[Web] Web Services Interoperability (WS-I) Organization. *Web Services Interoperability (WS-I)*. http://www.ws-i.org/.

[WK05] Huaigu Wu and Bettina Kemme. Fault-tolerance for Stateful Application Servers in the Presence of Advanced Transactions Patterns. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 95–108. IEEE Computer Society Press, 2005.

[WKM04] Huaigu Wu, Bettina Kemme, and Vance Maverick. Eager Replication for Stateful J2EE Servers. In *Proc. of the OTM Confederated Int. Conf. On the Move to Meaningful Internet Systems: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1376–1394. Springer Verlag, 2004.

[WM06] Dan Woods and Thomas Mattern. *Enterprise SOA: Designing IT for Business Innovation*. O'Reilly & Associates, Inc., 2006.

[WS92] Gerhard Weikum and Hans-J. Schek. and Applications of Multilevel Transactions and Open Nested Transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 516–553. Morgan Kaufmann Publishers Inc., 1992.

[WS97] Devashish Worah and Amit P. Sheth. Transactions in Transactional Workflows. In *Advanced Transaction Models and Architectures*, pages 3–34. Kluwer Academic Press, 1997.

[WSP+00] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database Replication Techniques: A Three Parameter Classification. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 206–215, 2000.

[WV01] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[YS05] Xinfeng Ye and Yilin Shen. A Middleware for Replicated Web Services. In *Proc. of the Int. Conf. on Web Services (ICWS)*, pages 631–638. IEEE Computer Society Press, 2005.

[ZMM02] Wenbing Zhao, Louise E. Moser, and P. M. Melliar-Smith. Unification of Replication and Transaction Processing in Three-Tier Architectures. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 290–, 2002.

[ZMM05] Wenbing Zhao, Louise E. Moser, and P. Michael 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.

The road to excess leads to the palace of wisdom. . . for we never know what is enough until we know what is more than enough.

– William Blake