Load-Balancing and Reconfiguration in a J2EE Application Server System

Xiaoguang Liang

Master of Science

School of Computer Science

McGill University

Montreal,Quebec

January 14, 2006

Your file Votre référence
ISBN: 978-0-494-24721-1
Our file Notre référence
ISBN: 978-0-494-24721-1

DEDICATION

I dedicate this thesis to my parents, who allow me to pursue my dreams.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Professor Bettina Kemme, for her guidance and encouragement. Whenever I had questions and difficulties in my thesis study, her thoughts and insights always inspired me. Second, I would like to thank Huaigu Wu, author of the Adapt SIB replication system, for his kind help. He is such a warm-hearted person and always provided me with useful information and suggestions. In addition, I would like to thank my husband Michel and my sons Leo and Olivier for their support to my study. Without their support, I would not be able to finish my thesis.

ABSTRACT

Distributed systems are the main architecture for enterprise applications. To develop a reliable distributed system, replication is necessary. The Adapt SIB replication system [27] provides a feasible solution for a reliable application server system. However, the system can not scale up since there is always only one primary server executing client requests. This thesis presents the LB system which is based on the Adapt SIB system, but with more functions. The LB system may have the same number of server replicas as in the Adapt SIB system, but it can have more than one primary server, each being able to execute client requests. Thus, a load-balancing mechanism is needed to distribute the load equally among different replicas. In addition, reconfiguration in case of failure, and restart must be considered as well. This thesis presents load-balancing and reconfiguration solutions for the LB application server system.

ABRÉGÉ

Les systèmes répartis sont l'architecture principale pour des applications d'entreprise. Pour développer un système réparti fiable, la redondance est nécessaire. Le système de redondance de Adapt SIB [27] fournit une solution réaliste pour un système fiable de serveur d'application. Cependant, le système ne peut pas prendre d'expansion puisqu'il y a toujours seulement serveur primaire exécutant des demandes de client. Cette thèse présente le système LB (Load Balancing) qui est basé sur le système Adapt SIB, mais avec plus de fonctions. Le système LB peut avoir le même nombre de serveurs que dans le système Adapt SIB, mais il peut avoir plus d'un serveur primaire, chacun qui peut exécuter des demandes de client. Ainsi, un mécanisme d'équilibrage de charge est nécessaire pour distribuer la charge également parmi différents serveurs. En outre, la reconfiguration en cas de pannes, et de redémarrage dans le système LB doivent être aussi bien considérés. Cette thèse présente des solutions d'équilibrage de charge et de reconfiguration pour le système de serveur d'application LB.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 Multi-tier architecture

Traditional enterprise applications were designed as all-in-one modules. User interface, processing logic, and database access were tightly coupled. Such systems are hard to design, maintain and modify. With the rapid development of networking technology, especially with the wide use of the Internet, the new generation of enterprise applications has a more feasible solution: multi-tier architecture.

A multi-tier architecture [8] separates an application into several layers: client layer, business logic layer, and data layer. The client layer contains user interfaces, the business logic layer implements business rules on the retrieved data, and the data layer represents the underlying database. Each layer can be implemented as a self-contained component and deployed onto a separate machine. Using a multi-tier architecture, each layer can be designed and maintained separately without affecting the functionality of the other layers. Another advantage of a multi-tier architecture is that the performance at each layer can be fine-tuned separately, hence providing better performance for the whole system. In other words, a multi-tier architecture makes distribution possible.

A very common architecture is depicted in Figure 1–1. The client layer only knows the middleware server, also called application server, sends all the requests to it, and gets the results back from the middleware server. The middleware server

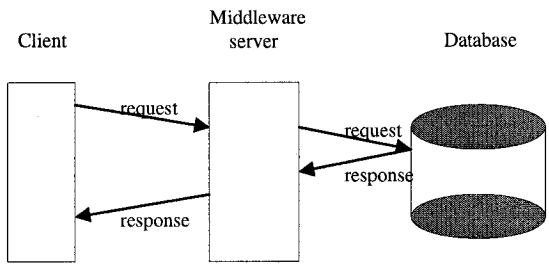


Figure 1–1: Multi-tier architecture

processes all the business logic. Important data that must be persistent, resides in the database, and the middleware server makes calls to the database whenever such data must be accessed. The middleware server also offers other functionalities, such as transaction management, database connection management, user session information management, etc. Hence, the middleware server plays a very important role in such a multi-tier system.

What will happen if the middleware server fails? Clients can not connect to the system, thus the enterprise application will not be available. Although the middleware server could come back to service soon after the failure, all information about currently connected clients is lost, which could cause data inconsistency in the database, and clients may be confused by the results. How can these kinds of disasters be avoided, that is, how can the system be made fault-tolerant? The solution is replication of the middleware server. Of course the database layer can also fail, and hence, affect the accessibility and correctness of the system. However, this thesis

focuses on the middleware tier, and we will only talk about the fault-tolerance of the middleware server.

1.2 Replication

Replication [7, 25] has been widely used in fault-tolerant systems for reliability purposes. Middleware server replication means there are several middleware servers in the system, and each of them is called a replica. Two kinds of replication schemes are often used: active replication and passive replication. Using active replication, the client sends a request to all the replicas, and all the replicas process the request simultaneously. Once there is a replica failure, the system is still available since all the other replicas are still alive. The advantage of this replication scheme is that failover, that is the period until the system is back to normal processing after a replica failure, is fast. On the other hand, each client request is processed at each replica, which means a significant overhead in the system during normal processing. Passive replication is depicted in Figure 1–2. Among many replicas, there is only one primary replica, which is accessible to the clients, and all other replicas are only backups of the primary replica. Only the primary replia executes the requests. The primary periodically sends its state information to all the backup replicas, and each backup replica will store the primary state information for backup purpose. If the primary fails, a backup takes over as new primary. There are two advantages of passive replication over active replication. First, it can be used when execution is non-deterministic because only one executes requests and the others simply apply the receiving state changes. Second, applying state information received from the primary has usually less overhead than executing the requests themselves. Hence,

the backups can be used for other computation puposes. But the coin always has two sides. Failover in passive replication is more complicated than in active replication. One of the backups will have to restore all the primary information and become the new primary in the system. The Adapt SIB replication system [27] developed by Huaigu Wu uses passive replication, which is the base of the LB system developed in this thesis.

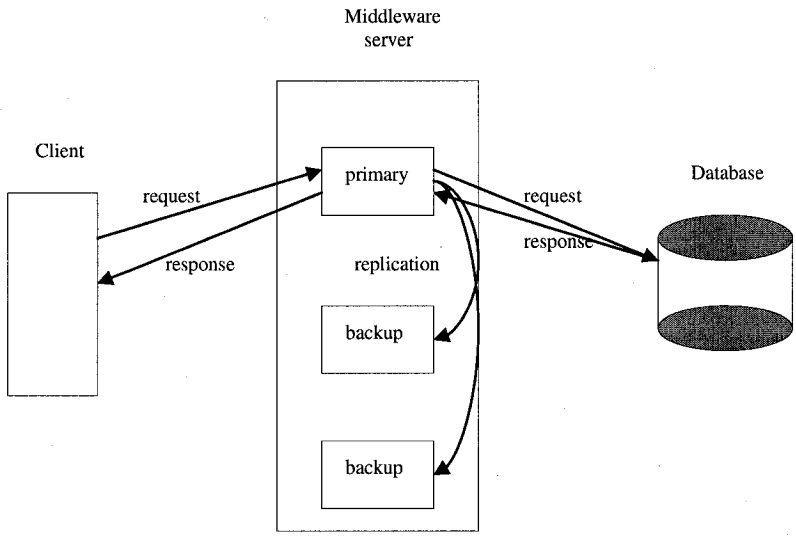


Figure 1–2: Passive replication of middleware server

1.3 Load balancing and reconfiguration

In passive replication, only one replica can be the primary while all the others are backups. Scalability can not be achieved by increasing the number of replicas. However, if we allow each replica to be primary for some clients and backup for some

of the other primaries, then the scalability of the whole system can be improved, as long as being a backup has less overhead than being a primary. This thesis proposes the LB system, an extension to the Adapt SIB replication system, that allows more than one primary. LB stands for load-balancing. The LB system distinguishes several replication groups, each having one primary and one or more backups. Each server replica can be primary for one replication group and backup for other groups. To distribute the workload to the different groups, we propose a load-balancing mechanism such that each replica gets an equal share of load. This approach provides both reliability and scalability.

The LB system, however, is also more complex to handle, especially in case of failure and recovery. What will happen to the LB system if a server replica fails? Several replication groups are affected, in particular, the group for which this server was primary. In addition, what will happen if the failed replica recovers? How can the machine be reintegrated into the system and possibly become a primary again? Failover and recovery are also called reconfiguration. This thesis proposes transparent and automatic reconfiguration mechanisms for the LB system.

The LB system has been implemented as an extension of the existing open-source application server JBoss [20, 21], which is based on the J2EE specification [22]. We have performed extensive experiments showing that scalability can be achieved by our load-balancing mechanism, and failure and recovery is handled smoothly, transparently, and with the least possible impact on the rest of the system.

The rest of the thesis is structured as follows. Chapter 2 provides backgroud information for the LB system. Chapter 3 describes in detail the Adapt SIB system,

which is the base of the LB system. Chapter 4 gives an overview of the LB system and its main features during normal processing. Chapter 5 brings reconfiguration issues of the LB system. In particular, it describes how the system is initialized, and what steps are taken when server replicas fail and recover. Chapter 6 presents the experiments and result analysis. Chapter 7 brings conclusion and proposes some future work for the LB system.

CHAPTER 2
Background Information

2.1 J2EE application server

J2EE [22] stands for JAVA 2 Platform Enterprise Edition, which defines a standard for distributed component-based applications. It aims to provide a maintainable, reliable and scalable platform for enterprise applications. A J2EE application server is an example of a middleware server as we mentioned in Section 1.1. But let us first talk about Enterprise Java Beans (EJB) [23], which build the programmable units for J2EE application servers. Basically, there are two kinds of EJBs: session beans and entity beans. Session beans are used to implement business logic (for example, a program that implements a money transfer or that keeps track of all the goods a user has selectecd for purchase while she or he is logged into an online store). There are two types of session beans: stateful and stateless beans. A stateful session bean is usually associated with a user session and keeps the information for this particular user during the period the user is connected to the system. A stateless session bean is used to perform arbitrary tasks, but will not keep state information once the task is finished. An entity bean is a representative of the underlying persistent data, for instance, it can be used to represent a data record of a database table. Entity beans are also stateful. The state of the entity bean must be synchronized with the data state in the underlying database. Each EJB has a home interface and a remote interface, the home interface allows a client to create, find, and remove

EJB objects; the remote interface is used for remote client access, and contains the business methods. Servlet and JSP are programmable units for presentation logic and control flow. They allow for an easy generation of web pages.

A J2EE application server consists of a Servlet container and an EJB container. The Servlet container serves as a runtime environment for Servlets and JSPs, which are mostly used to implement presentation logic. The EJB container provides runtime support for EJBs, where the business logic is processed.

The J2EE application server offers several other services as well, such as transaction management service, concurrency control service, and database connection (JDBC) service. These services can be used by the EJB applications. The J2EE application server makes EJB application development easier since the application does not need to implement those services that the J2EE application server provides, but can simply call them when needed.

A transaction is a set of operations logically belonging together. A transaction has change state in the application server (stateful session beans) and the database (represented by entity beans). The transaction manager (TM) of the application server manages all the transactions. Once a *begin* transaction operation is received by the transaction manager, it starts a new transaction and assigns this transaction a transaction number. All the operations with this transaction number are executed as part of the transaction until a *commit* or *abort* operation is encountered. If the operation is *commit*, the transaction manger will make the changes on entity beans of this transaction permanent (i.e., commit the database transaction). If the operation is *abort*, the transaction manager will undo all the operations executed so far (and

the database transaction aborts). There are two ways of transaction management in a J2EE application server: bean-managed (BMT) or container-managed (CMT). BMT means that the EJB application developers have to manage the transaction programmatically, that is, the programmer has to indicate in the code where the transaction should start and where it should end. CMT means the EJB container will take care of the transaction management. The transaction boundaries can be set up in a configuration file. When a EJB is deployed in the EJB container, a transaction attribute is specified for each method of the bean in the configuration file. The container reads the necessary information from the configuration file and manages the transaction accordingly. For example, assume the transaction attribute for a method is set to *Required*. Whenever the method is called, if the calling thread is associated with a transaction, the actions within the method become part of this transaction. If no transaction is associated, the container submits a *begin* transaction request to the transaction manager, then the method executes within this transaction. After execution finishes, the container calls the *commit* operation to terminate the transaction.

J2EE application servers are now widely used in enterprise application systems. Their reliability and availability are crucial to the whole enterprise application system. Many commercial J2EE application server providers start to provide some replication mechanisms to increase reliability.

2.2 Group communication systems

Group communication systems [24], as implied by their name, provide communication for all members of an application group. They provide a number of

messaging services to applications, and make reliable distributed systems possible. Some messaging services offered by group communication are as follows:

Reliable multicast and message ordering: A group communication system provides reliable multicast from senders to all receivers of the same group. Multicasting happens when a member of an application group needs to send a message to all its members of the same group. The reliability of group communication guarantees that 1) each member receives a message at most once 2) if a member multicasts a message m then it will eventually receive m unless it crashes and 3) when a member p receives a message, and if p does not fail for sufficiently long time, all other members in the same group will receive the message unless they fail. A stronger guarantee, called uniform reliable delivery, guarantees that if a member receives a message, even if it fails immediately after, all other members in the same group will receive the message unless they fail. In addition, group communication systems provide message ordering mechanisms. In this thesis, we are interested in **1) FIFO ordering:** if a member sends a message m before it sends message m', all members in the same group receive m before m' **2) Total ordering:** if two members receive m and m', both either receive m before m' or m' before m.

Membership services, also called view management: Group configuration can change if group members join or leave the group. A view reflects the current membership of a group. When a group member joins the group, all the members of the group (including the new one) will receive a new view, which includes all the members in the current group. When a group member leaves the group, all the remaining members of the group will also receive a view change event, in which the

member left is taken out of the new view configuration. The group communication mechanism assures that all the members of a group get the view change events in the same order. View management of group communication allows easy management of group members. This provides a very useful tool for upper level application designers to manage the membership of an application group. For example, a member p has received as its last view the view V. Assume a new view change event happens in the group, and the member receives a new view change event V'. If there is a member q in V' which was not in view V, that implies a new member joined the group. The member p could decide accordingly what it wants to do with the new member q, for example, send it the latest state information for recovery purposes. If there is a member q in V that is no more in V', that implies a member left the group by failure or voluntarily. Hence p may do something accordingly as well. Examples of group communication systems are Horus [17], ISIS [6], Transis [9], Totem [16] and Spread [14]. Spread is an open-source group communication system, which we use in our implementation.

2.3 JBoss application server

J2EE has provided a standard for building application servers. There are many commercial application servers in the market already, such as IBM WebSphere, BEA WebLogic, etc. Besides these commercial products, there are also several open-source application server products available. JBoss is one of them.

JBoss [20, 21] provides a full J2EE implementation, containing an EJB container and a Servlet container, and providing services like JTA (Java Transaction), JDBC (Database connectivity), JNDI (Java Naming Directory Inteface), etc. It offers all

of these services with the help of JMX (Java Management Extension), which serves as a spine for integration of all the modules, containers, and plug-ins.

The JBoss EJB container is the core implementation of the JBoss server. Here we have to mention *EJBObject* and *EJBHome*. *EJBObject* is the implementation of a bean's remote interface, and *EJBHome* implements a bean's home interface. When a client wants to access an EJB object for the first time, it first gets a reference to *EJBHome* from the EJB container by looking it up in the JBoss naming server. Then the client calls the Create() method on *EJBHome* and gets back an *EJBObject* reference (RMI stub). Once the client has the *EJBObject* reference, it can call the method of this EJB directly through the EJB container.

JBoss uses a smart interceptor stack to wrap the services for EJB access [18]. A client call to an EJB component will go from one interceptor to another until it reaches the component it calls. Upon completion of the call, the result will be returned back through the interceptor chain in reverse order until it reaches back to the client. The interceptor architecture of JBoss is shown in Figure 2–1.

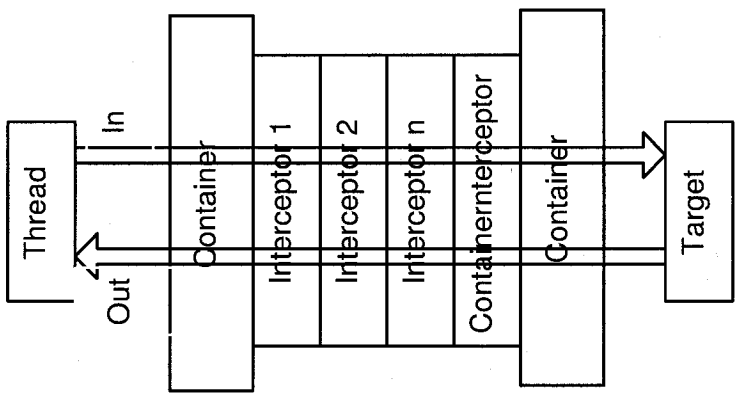


Figure 2–1: JBoss interceptor architecture

When the client call is forwarded to an interceptor, the interceptor can do some extra work before the call is forwarded to the next interceptor and after the response is received by this interceptor. It provides a third party the possibility to put its own interceptor between two existing interceptors. This opens an easy door for a replication framework.

2.4 The Adapt framework for replication

Replication of a J2EE application server provides reliability for enterprise application systems. There exist many replication algorithms, and the study of new replication algorithms is still ongoing. To compare replication algorithms, they have to be implemented into the J2EE application server. If each of them was implemented independently and embedded into the application server, this would require a lot of work for each implementation. The purpose of the Adapt framework [4, 15] for replication is to facilitate the implementation of replication algorithms. It provides a uniform abstract framework such that a replication algorithm only needs to use the API of the framework, and the framework will then deal with the implementation issues of the underlying application server.

The Adapt framework inserts interceptors into the interceptor stack of the JBoss server, which behave like component monitors of client calls. Once a client call comes to one of the inserted interceptor, it is directed to a replication algorithm implementation which does the corresponding replication work. The interceptor then redirects the client call to the next interceptor. In this way, the framework does not change the underlying application server design and implementation. It provides an API such that a replication algorithm can be easily deployed into the server. The

Adapt framework for replication has been developed for the free open-source JBoss server. Figure 2–2 shows how the Adapt framework intercepts client requests. *1* and *2* in Figure 2–2 are the possible interception points during an invocation between two components. *1* can be used to implement some client-side logic for the server replication algorithm. For instance, it can resend the request to a new server replica in case of failure. *2* is at the server side, where replication can take place before the call is forwarded to the callee.

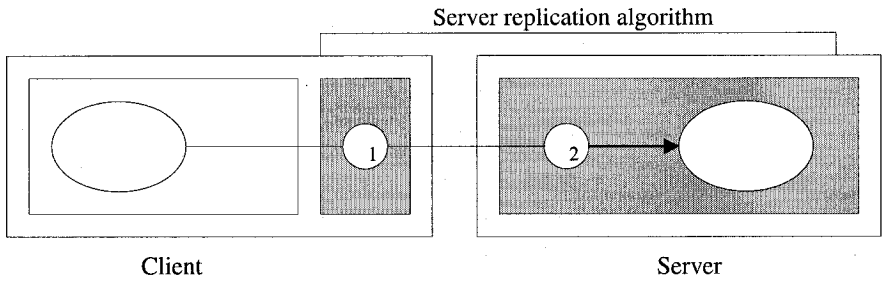


Figure 2–2: Interception of invocations in the Adapt framework

2.5 Cluster

A cluster is a group of computers connected together by a local area network which work together as one machine from the view point of the client. Clients of the cluster know the address of the system, but have no idea whether the system is one machine or a cluster. A cluster can provide high availability, high reliability and high scalability.

2.6 Load balancing

If there is more than one server in a cluster which can handle client requests, it is quite clear that we need to have a mechanism to allow every such server to get its turn to serve the clients. Load balancing [1, 2, 10, 12, 19] is a mechanism to distribute the tasks among all available servers such that each server gets an equal share of work, thus improving the scalability of the cluster. For example, [10] suggests *Random*, *Round Robin*, and *Load* load-balancing algorithms.

- `Random` : is a simple load-balancing algorithm. When a new task arrives for execution, one of the servers is picked up randomly. The advantage of this load-balancing algorithm is that it is simple to implement. The disadvantage is that the same server may be picked up constantly such that this server is heavily overloaded while other servers rest idle.
- `Round Robin` : is another simple load-balancing algorithm. A server is chosen using the round robin method. Each server gets its turn one after another. The advantage of this load-balancing algorithm is also its simplicity. The hope is that the tasks are more evenly distributed among all available servers compared to the random load-balancing algorithm. However, if different tasks have different load associated to them, the load will not be distributed equally across the servers.
- `Load` : is a more complicated load-balancing algorithm. In this load-balancing algorithm, each server has a load, which could be measured by CPU usage, number of current requests executing, etc. Each server gets its task share according to its load at that time. The higher the load, the less likely that it

will get a new task assigned. The advantage of this load-balancing algorithm is that the load can be more evenly distributed across all available servers. The mechanism is more suitable than the *Round Robin* if tasks have different loads. The disadvantage is that it is more complicated to implement. For example, it is difficult to make an accurate measurement of the current system load.

2.7 Related work

e-Transaction [11] is a replication protocol used in three-tier systems to provide reliability of the middleware server. When a client request comes to a server replica, it is executed within the context of a transaction. Additionally, a request marker is inserted into a marker table in the database. If the transaction is committed, the marker becomes permanent in the database. If the transaction is aborted, the marker is automatically removed by the database together with all other effects of the transaction. In case of failure of this server replica before it sends a response to the client, the client may submit the same request again to another server replica. The new replica checks the marker table first to see if the request has already been executed successfully. If it finds the marker associated with this request, then the response is sent directly to the client. Otherwise, the request is treated as a new request. This is also called exactly-once guarantee. Since a request never leads to more than one execution, we have at-most-once execution. Since the client resubmits a request when it does not receive a response, we have at-least-once. In total, this yields exactly-once execution. e-Transaction does not consider stateful middleware servers (as J2EE with its stateful session beans). Neither does it discuss load-balancing.

WebLogic application server [5]: Passive replication is used by WebLogic, and only the primary server processes client requests. The server state replication from the primary to the backup happens after the response is returned to the client. In case that the primary fails after it returns the response to the client but before it sends the state information to the backup, the new server state is lost at the backup. In this case, once the backup becomes the new primary, the server state will not be consistent.

JBoss application server [13, 20, 21]: Passive replication is also used by JBoss. Different to WebLogic, replication happens at the end of each request just before returning the response. If the transaction associated with this request is eventually aborted, the server state of the backup will not be consistent with the server state of the primary. Also, if the primary crashes in the middle of executing a request, the backup has inconsistent state.

Both WebLogic and JBoss provide cluster support with load-balancing. However, their replication algorithms do not really provide complete fault-tolerance in some cases as we mentioned above.

The Adapt SIB Replication System [26, 27]: This scheme uses passive replication and sends state changes before the *commit* of transactions. It considers sateful servers and coordinates execution to provide data consistency across middleware server replicas and database. Additionaly, it provides exactly-once execution similar to e-Transaction. The Adapt SIB system is actually a cluster system, but because there is only one server in the cluster that works as a primary while all others only work as backups of the primary, the Adapt SIB system is a restricted

case of cluster system, which only provides availability and reliability. The goal of this thesis is to extend SIB to a scalable cluster system, which is called LB system.

CHAPTER 3
The Adapt SIB Replication System

In this chapter, we talk about the Adapt SIB replication system [26, 27]. The Adapt SIB system serves as the base of the LB system, which will be discussed later. The main assumption in the Adapt SIB algorithm is that each client request generates exactly one transaction in the application server. That is, the execution of a client request r happens in the context of one individual transaction t. Hence there is a 1-1 association between transaction and request.

3.1 Correct replication of J2EE application servers

There are two things the replication algorithm has to guarantee in order to achieve fault-tolerance of a stateful J2EE application server. The first one is to guarantee the state consistency between the replicated application server and the backend database. State consistency means that if a transaction changes both the state of the application server and the database, the state of the application server and the state of the database are consistent after the transaction is committed (both have the state changes associated with the transaction) or aborted (none of the state changes remains at application server or database). Without replication and assuming no failures, this state consistency is guaranteed by the transaction mechanism. But with replication, the state consistency among all the replicas of the application server must be guaranteed such that in case of failure, the backup replica can become

the new primary without losing state consistency. Another requirement is to guarantee that a client request is always executed exactly once. For a given request in case of failure of the primary, the new primary must be able to determine if the request was already executed and the corresponding transaction committed. If yes, it will not execute it again. Otherwise, if a client had sumitted a request but the primary crashed before finishing execution and returning a response, hence the corresponding transaction aborted, the request should be executed again by the new primary. State consistency and exactly-once execution are critical for the correctness of a replication algorithm. A good replication algorithm should always satisfy these two guarantees.

3.2 Simple algorithm

A standard passive replication algorithm would do the following steps. When a request arrives from the client at the primary, a transaction is started and the execution can change one or more beans in the application server and access the database. At the end of execution, the primary sends all server state changes to the backups, then it commits the transaction, and then it returns the response to the client. For simplicity, let's ignore transaction aborts due to deadlock or application semantics (e.g. not enough money in account). That is, let's assume that each request leads to a successfully committed transaction as long as there are no failures (the real Adapt SIB algorithm considers aborts). However, this is not enough to gurantee state consistency and exactly-once execution. Let's have a look at some failure cases and the actions the backup that becomes the new primary has to perform after the crash as part of the failover procedure.

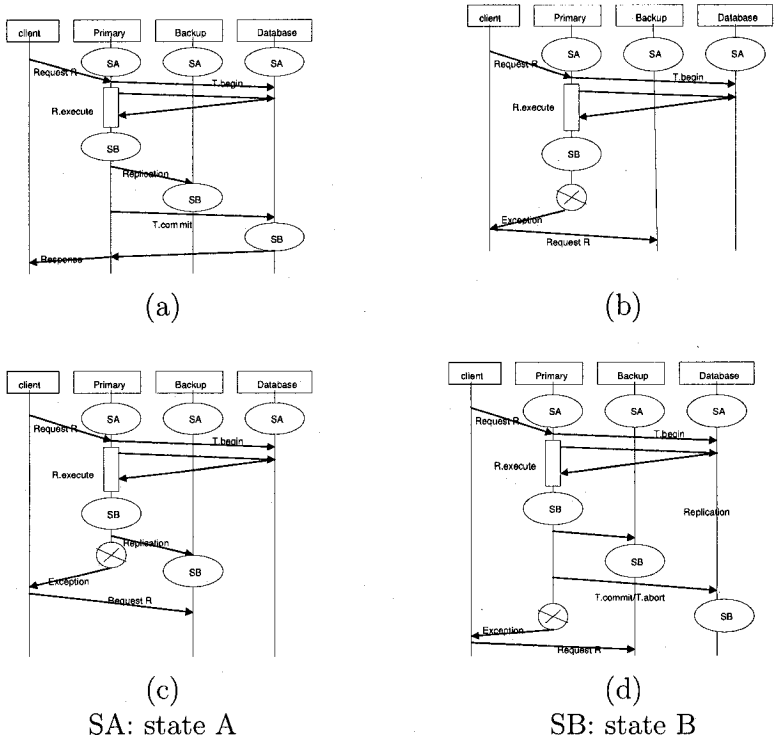


Figure 3–1: Failure cases

3.3 Analyzing failure cases and the actions on the backup

Figure 3–1 (a)-(d) show diagrams of possible execution scenerios for request R. We see the actions executed at the client, the primary, the backup, and the database.

In Figure 3–1 (a), the replication algorithm works in its normal processing state, there is no failure, and the state of the primary replica (state is SB) is consistent with the state of database (state is SB) after the transaction associated with request R commits; the backup replica also keeps the right version of replication information (state is SB). State consistency is guaranteed.

In Figure 3–1 (*b*), the primary replica fails before it sends the state change information to backup replicas and commits the transaction. At the crash of the primary, the database will abort the above transaction associated with request *R*. Although the update of the state of the primary replica is lost, the backup replica (state is SA) and database (state is SA) are consistent at this point since none of them has state changes associated with request *R*. However, execution is at-most-once. To achieve the exactly-once execution, either we rely on the client to resubmit the request or we have such automated resubmission system in place. In the Adapt framework, there is an interceptor point at the client side that allows for automated resubmission. If the backup becomes the new primary and the client resends the same request, the request can be processed normally on the new primary. Exactly-once is guaranteed.

In Figure 3–1 (*c*), the primary replica fails after updating the backup's state but before terminating the transaction related to request R. The backup received the update information and it has a new state (state is SB), but the database aborts the transaction at the time of the crash and does not contain R's updates (state is SA). Now, if the backup becomes the new primary, it is obvious that the state of the application server (state is SB) is inconsistent with the state of the database (state is SA). There must be a control mechanism to solve this inconsistency problem. The new primary has to discard the state changes received from the old primary. Then it should re-execute the client request upon resubmission.

In Figure 3–1 (*d*), the primary replica fails after it terminates the transaction at the database, but before it returns the response to the client. The new primary (with

state SB) may not re-execute the request upon resubmission but should immediately return the result.

A problem is how the new primary distinguishes the cases depicted in Figure 3–1 (*c*) and 3–1 (*d*). In both cases it has received the state changes and the client will resubmit the request. The new primary must be able to detect whether the corresponding database transaction committed before the crash of the old primary or aborted at the time of the crash.

3.4 Providing state consistency and exactly-once execution

From the above analysis, we can see how important it is to solve the state consistency problem and the exactly-once problem. Now let us take a closer look how the Adapt SIB replication system solves these two problems.

- `Marker insertion` : We keep a marker table in the database to store committed transaction ids. If a transaction updates the database, a marker is also inserted into the marker table. Since the marker insertion is an operation of this transaction, if the transaction is eventually committed, then the marker is permanent in the marker table. If the transaction is aborted, then the marker is removed from the marker table as part of the abort. This marker insertion is critical to guarantee correctness.
- `Keeping track of responses` : We keep a Hashtable *RR* with requests and corresponding responses as additional state at each replica. When the primary sends the state information on behalf of a client request to the backups, it also sends the response it is going to return to the client.

- `Transparent resubmission of outstanding requests` : We have an interceptor at the client side, which intecepts the client request and also the response that the server returns to the client. If this interceptor receives an exception from the server, then it resends the client request again to the new primary server until it gets back a response to the client request. This whole procedure is completely transparent to the client.

Let us look again at the example execution. For the scenarios in both Figure 3–1 (a) and Figure 3–1 (b), the state consistency and exactly-once execution are guaranteed with or without marker insertion and Hashtable *RR*. In Figure 3–1 (a), the client will not resubmit the request. In Figure 3–1 (b), both backup and database have no information regarding the request, and it is safe to re-execute it when the client resubmits it.

For the scenario in Figure 3–1 (c), neither state consistency nor exactly-once is guaranteed without the help of marker insertion and *RR*. The primary fails before updating the database, which means there is no marker for this transaction in the marker table. During failover, the backup will check the marker table in the database for all transactions for which it received state change message. If a transaction id can not be found in the marker table, then the replication information of this transaction is dropped at the new primary to provide state consistency. If the client resends the same request again, it will become a new request to the new primary, providing exactly-once execution.

For the scenario in Figure 3–1 (d), the marker for this transaction is permanent in the marker table. During the failover time of the backup replica, after checking in

the marker table, the backup knows that the transaction actually committed in the database. It will keep the replication information for this transaction and include the response in the hashtable RR. After the failover, the new primary has state consistent with the database. In this case, once the client resends the same request to the new primary, the new primary will first check its Hasthable *RR*, find the response, and return it right away to the client, without re-execution, guaranteeing exactly-once execution.

3.5 Protocol details of the Adapt SIB replication system

As we mentioned above, in the Adapt SIB replication system, there is more than one application server, but only one of them can become the primary to process client requests, all the other replicas only store the replication information sent by the primary replica. The system is implemented using the Adapt framework for replication based on JBoss application server. The group communication used in the system is JBORA [3], which is a JAVA API of the group communication system Spread [14].

The Adapt SIB replication protocol can be separated into two parts: server-side protocol and client-side protocol. In this section we will describe both protocols in detail.

3.5.1 Assumptions for the Adapt SIB replication system

1. A replica of the system always works correcctly until it crashes. The primary is always available until it crashes.
2. The database server is always reliable, no database failure occurs.
3. The client does not fail.

4. We assume no network partition occurs.

3.5.2 Some terminology used in the Adapt SIB replication system

1. Client Replication Manager (CRM) implements the replication algorithm of an EJB at the client side. Each EJB has its own CRM, which is an RMI stub and is sent back by the server when the client calls Create() on *EJBHome*. Inside CRM, there is a serverlist of the replicas in the replication group. The CRM is the implementation of the interceptor *1* at the client side as depicted in Figure 2–2. All the requests of the client to this EJB will always go through this CRM first.

2. Replication Manager (RM) is actually the component monitor described in Section 2.4, which implements the replication algorithm at the server side. Each replica has an RM. The RM on the primary first intercepts all requests (requests to EJBs or *begin/commit* requests to the transaction manager) and does some pre-request processing before sending the request to the required EJB or the transaction manager. When it gets back the response from the transaction manager or EJB it will do post-request processing, e.g., sending the replication information to all backup replicas. Then it returns the response to the client, actually the CRM at the client side. Each RM keeps a status variable to keep track of its state. At the initialization of the system, every replica has a status of *pre-primary*, which means any of them can become a primary. Once the primary is decided in the system (the one who first receives a client request becomes the real primary), all other replicas set their status as *backup*. Once the primary replica fails, a new primary will be elected from all the backups, and it will do the failover and its status becomes *during_failover*. Once the failover finishes, it become the new primary. If during normal processing, a new

replica joins the replication group, the new replica has a status of *during_recovery*, and after the recovery finishes, it has a status of *backup*.

3.5.3 Client side replication protocol

The client is not aware of the CRMs. Once the client sends a request to the application server, the CRM of the EJB intercepts the request, and the CRM will forward the request to the primary server. The server then processes the request, and sends back the response first to this CRM, and the CRM will check the response to see if the response is normal. If the response is normal, the CRM will then return the response to the client. If the CRM gets an exception from the server, that means the primary replica has failed, the CRM will check its serverlist, connect to the next available server, and asks whether it is the new primary. Once it has found a new primary replica, it resends the request to the new primary. This repeats until a correct response is received.

3.5.4 Server side replication protocol

The sever side replication protocol includes four parts: primary protocol during normal processing on the primary replica, backup protocol during normal processing on the backup replica, new primary protocol at the time of failover on the new primary replica, and recovery protocol when a failed replica recovers or a new replica joins the replica group.

In each RM, there are some attributes implemented:

- `Hashtable RR` : stores the pairs of client request and its response.

- `Hashtable TRS` : stores information for transactions that are currently executing, which are the pairs of transaction and all the EJB instances that the transaction accessed plus the response for the client if already existing.
- `Vector ES` : stores the EJB state information for each committed transaction.
- `LinkedList MB` : message buffer to store the replication messages sent by the primary RM.
- `Boolean is_Failover` : If it is true, the RM is during failover.
- `Boolean is_Recovery` : If it is true, the RM is during recovery.

Primary protocol during normal processing

The primary RM has two parts of work to do: the transaction interceptor intercepts a *begin/commit* transaction request before the request reaches the transaction manager of the normal JBoss server. The request interceptor intercepts requests for EJBs, and eliminates duplicate requests.

In the request interceptor, once a client request to execute a method on an EJB comes to the primary RM, it first checks its *RR* table to see if there is already an entry for this request. If yes, which means that it is a duplicate request, the response stored in the *RR* will be returned to the client right away. If no, which means this is a new request from the client, the RM will hand it to the required EJB, and once the RM gets back the response, it will store it in *RR* and then return it to the client. The EJB called by the client can make calls to other EJBs as well, which are nested calls. All the nested calls belong to the same transaction, and all the EJB state changes are recorded in *TRS*.

In transaction interceptor, once a *begin/commit* request of a transaction is intercepted, it will do some replication work first, and then forward the operation of the transaction to the real transaction manager of JBoss server.

If the request is a *begin* transaction, the primary RM will make a new entry into *TRS* with a new transaction id assigned to this new transaction.

If the operation is *commit*, the primary RM inserts a marker into the marker table in case that the database was updated. In order to decide whether the database was updated, the transaction interceptor also intercepts every request to the database and decides whether the access was read or write. The RM also generates a replication message and multicasts it to all backup replicas in the current group. Using uniform reliable delivery, the replication message includes the transaction id, the pair of the request and the response, and all stateful EJBs accessed in this transaction, including the state of each stateful session bean, but not the state of any entity bean since its state is always synchronized with the database. Then the primary RM waits until it receives its own replication message (note that with total order multicast, it receive its own message). That guarantees that the backups have received the message. Then the *commit* is forwarded to the transaction manager of JBoss who finally commits the transaction in the database. The commit confirmation of the transaction manager is again intercepted. The RM generates a committed message and multicasts it to all backups to confirm the replication information sent before.

Backup protocol during normal processing

The message buffer *MB* of a backup stores all the messages sent by the primary RM. The message processor associated with this backup RM will constantly try

to retrieve a message from the *MB* in the order they were received and process it accordingly. If the message processed is a replication message, then the related information of this transaction will be put into *TRS* instead of *ES* because it is still possible that the transaction aborts due to crash. Once the committed message is processed, the information of the transaction will then be removed from *TRS* and put into *ES* and *RR*.

New primary protocol at the time of failover

Once the primary RM fails, one of the backup RMs will be chosen to become the new primary RM. But before the backup can really become the new primary, it must pass a failover period. The new primary can be sure that if a transaction is committed in the database, it must also have been committed in the primary, and if a transaction is aborted in the database, the primary crashed while the transaction was still alive.

Once the backup detects the failure of the primary, it will set its *is_Failover* to true, and it will trigger its failover process. First it processes all the transactions in *TRS*, that is, it received the replication message but not the committed message. It checks the marker table in the database to know if the transaction is committed in the database or not. If a marker is found in the marker table, which means the transaction is already committed in the database, the RM will confirm the transaction by putting replication information into *ES* and *RR*. Otherwise, if no marker is found in the database, which means the transaction is actually aborted in the database, the RM will remove the replication information of this transaction from *TRS* and drop it. In either case, the state of the RM and the state of the database

is always consistent. However, the failover is not finished yet, the new primary RM must reconstruct each stateful session bean (SFSB) and entity bean (EB) involved in the committed transactions. The SFSBs can be constructed with the replication information in *ES*. The EBs will be constructed by retrieving their state from the database since their state is always consistent with the state of the database. Now the failover of the new primary has finally finished. The indicator *is_Failover* is set to false. The new primary now is ready to handle client requests.

Recovery protocol when a failed replica (or a new replica) joins the replica group

A failed replica may want to join the group again. Once a replica fails, it loses all its state information, and becomes a brand new replica to the group. It is assumed that the recovered replica can only become a backup replica. The main task here for the recovery is to make the new recovered replica to have a state consistent with the state of all the other backups in the system. There are two problems to be solved. First, who will send the recovery information? Second, how to make sure that the state of the new replica is consistent with the state of all the other backups?

Once a new replica joins the current group, all replicas in the system will detect it due to the view change event delivered by the GCS (Group Communication System). All backups in the system then multicast a *query sender* message. The *query sender* message is sent using total order multicast. With this all the backups receive the same first *query sender* message. The backup who sent the first *query sender* message then becomes the peer site to send the recovery information to the new joined replica. Upon reception of the first *query sender* message, the sender of this *query sender* message generates the recovery message. The *TRS, RR, ES* will be locked in order

to coordinate recovery with the reception of replication messages from the primary. All information received before the *query sender* message will be transferred to the new site. All messages received after the *query sender* message will be processed by the new site itself.

At the new replica, its indicator *is_Recovery* is set to true. It also receives the *query sender* message from all the backups. Once it receives the first *query sender* message, it knows who the peer site is, and then it will also lock the *TRS*, *RR*, *ES* such that its message processor will not be able to perform message processing. Once the recovery information is received, the new replica first parses the recovery information, and then clears the *MB* according to the parsed recovery information to drop the messages that are already covered in the recovery message such that the state of the new replica is consistent with the state the peer site sends. Once the recovery is finished, the *is_Recovery* is set to false, and the new replica becomes a new backup of the system. It can start to process the replication messages as a normal backup.

CHAPTER 4
LB System During Normal Processing

4.1 Design concept of the LB system

The Adapt SIB system has one replication group with one primary and several backups. In the LB system, we have several such replication groups, each with a primary and several backups. Each primary is able to handle client requests. Since there is more than one replication group in the system, we must have a load-balancing algorithm to dispatch a client to one of the replication groups, such that each replication group gets its share of the whole workload of the LB system. But which server in the system will do the load-balancing work? A first solution is that we have a dedicated server, which works only as a load-balancer, also called dispatcher. This architecture is quite simple and easy to implement, but what will happen if the dispatcher fails? The whole system will be unavailable. Another solution is that we have a load-balancer group similar to the replication groups. Each LB server has a load-balancer which is a member of this group. But only one of them works as a primary load-balancer, all the others are only backup load-balancers. If the primary load-balancer fails, one of the backups takes over and becomes the new primary load-balancer. Using a group of load-balancers has the advantage that if the primary fails, the GCS automatically detects the failure and can inform the backups. The group concept also helps the load-balancers to communicate with each other to exchange enough information such that each of them has the same up-to-date information of

the whole system for dispatching purposes. We use the second solution in our LB system.

4.2 General architecture

Each site (machine) runs one LB server. In the following, we identify an LB server by the site name. An LB server consists of several active components. It has one *LoadBalancer (LB)*. The LBs running on the different servers belong to one group, which we call the *LB group*. One of the LBs in this group works as a special primary LB. It is responsible for executing the load-balancing algorithm and dispatching clients to replication groups. The other LBs serve as backups for the primary LB. We refer to the LB server with the primary LB as the primary LB server.

Furthermore, each LB server can have several *ReplicationManagers (RM)*. Each RM of an LB server is a member of a different replication group, and at most one RM of an LB server is the primary RM of its group. All the others must then be backups in their corresponding groups. An RM in the LB system is not exactly the same as in the Adapt SIB system. In the LB system, each group still has one primary RM, and some backup RMs. But the primary RM will not intercept client requests directly, but through the local LB. The LB of an LB server S receives the client requests sent to S and forwards them to the primary RM of S. The primary RM takes a request from the LB, performs its actions according to the Adapt SIB algorithm, and returns the result to the LB which forwards it to the client.

Note that the different LBs and RMs are identified via their server/site name in their groups. For instance, the LB of LB server S is identified as S in the LB group,

and an RM (primary or backup) for group G running on S is also identified as S in replication group G. In the following, when we say an LB server fails, we assume an LB server entirely crashes, including its LB and all its RMs crash.

4.3 Load Balancing during normal processing

In the LB system, the load-balancing algorithm works at the client level, which means once a client is assigned to a replication group, all its requests will go to that group until the client disconnects or the primary of this group crashes. That is, a client session is scheduled to a replication group. In this way, we can assure that all the information about the same client session is always stored in the same replication group. If a client first sends a request to group $G1$, and then it sends its next request to group $G2$, neither group $G1$ nor group $G2$ has the full information about this client session, which is not desired in a stateful application system.

Let's now have a closer look at how clients are assigned to groups during normal processing without crashes. As we already know there are many load-balancing algorithms available, such as *Random*, *Round Robin*, *Load*, etc. It was not the purpose of this thesis to invent a new load-balancing algorithm. Instead, this thesis focuses on a framework that allows dynamic reconfiguration of a cluster-based system, in which each server can handle client requests and at the same time serves as backups for other servers. Hence, we designed our framework such that any load-balancing algorithm can be plugged into the framework. Our current system is implemented with the *Round Robin* algorithm. In the following, we discuss how this new framework intercepts client requests and assigns new clients to a replication group.

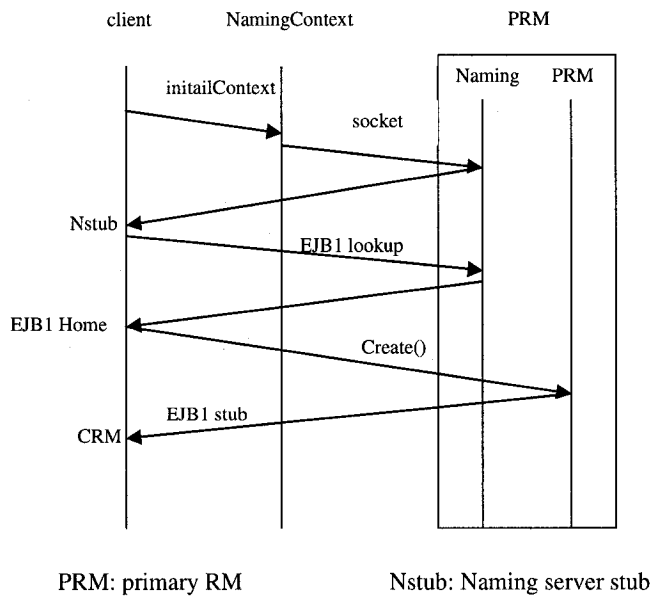


Figure 4–1: Without load-balancing

Figure 4–1 shows how a client connects to the application server in the Adapt SIB system, which only has one replication group, and no LB group. The client first calls *initialContext*, which is a client side agent for the naming server on the server side. *initialContext* has access to a list of servers called *serverlist*, which contains the server names that host an RM (primary or backup) of the replication group. The *initialContext* then connects to the naming server on the server with the primary RM to get back a naming server stub. Then the client makes a lookup call on the required EJB via the naming server stub, and gets back the *EJBHome* object. Once the client has the *EJBHome*, it calls *Create()* on this object to get the remote interface of this

EJB, and at the same time, a *ClientReplicationManager* (CRM) of this EJB is also downloaded to the client side, which is part of the Adapt SIB replication algorithm.

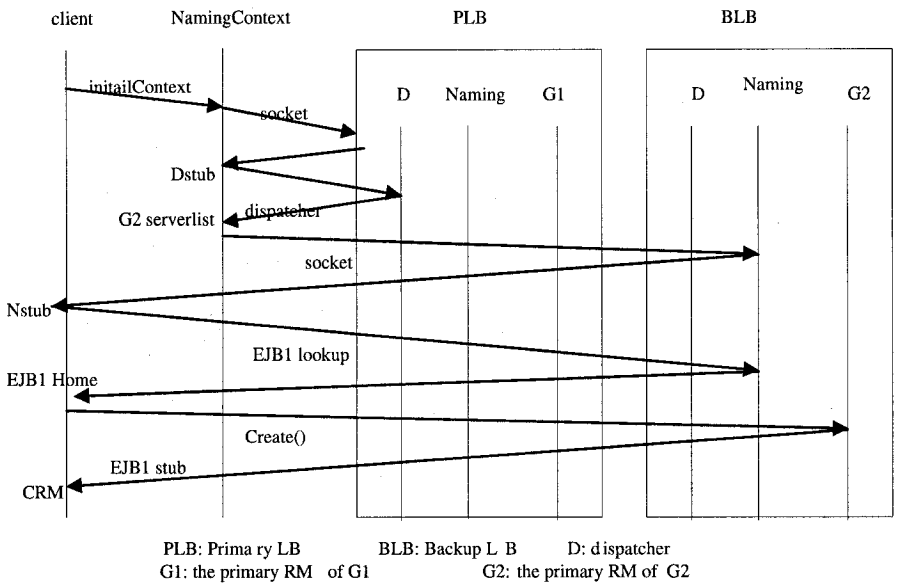


Figure 4–2: With load-balancing

Figure 4–2 shows how the load-balancing algorithm works in the LB system. *initialContext* now has a list of servers that run an LB. When a client calls *initialContext* for the first time, the *initialContext* in the LB system does something more than it does in the Adapt SIB replication system. It contacts the primary LB server using a socket connection to get the LB stub back, then accesses the primary LB via the LB stub. The primary LB chooses the replication group according to the load-balancing algorithm configured in the LB configuration file and returns the server list of this replication group to the *initialContext* of the client. In Figure 4–2, this is *G2*. This is the replication group that this client will make its requests to.

Thus, the load-balancing for this client is completed. If the *initialContext* connects to a backup LB server instead of a primary LB server, the backup LB server will send a null object back instead of an LB stub. In this case, the *initialContext* will try the next LB server on its server list until it finds the primary LB server. Once the *initalContext* gets the server list of the assigned replication group, it will then connect to the LB server with the primary RM of the group that was assigned to the client. From there nearly the same steps are taken as in the Adapt SIB replication group as shown in Figure 4–1. The only difference is that, as mentioned before, all further requests of this client actually go through the local LB of this LB server. This LB does some extra actions needed for fault-tolerance. We will discuss these actions later.

CHAPTER 5
Reconfiguration

While execution during normal processing is a rather straightforward extension of the original Adapt SIB system, dynamic reconfiguration is more challenging. There are three different situations in which reconfiguration takes place. First, at system startup one LB server after the other is started and joins the system. Second, when an LB server crashes, it leaves the running system. Third, when a crashed LB server restarts after a crash, it rejoins the system. In all three cases, the LB group and the replication groups associated with the leaving/joining LB server are affected. That is, when an LB server joins/rejoins the system, it joins the LB group and all replication groups for which it has an RM configured to be running. If an LB server crashes, it leaves all such groups. Information about the group configuration, of who is primary and who is backup must be adjusted during a reconfiguration so that after reconfiguration has finished all LBs and RMs know the current state of the system.

In the following, we use a stepwise approach to describe the reconfiguration process. We first introduce terms and data structures that play an important part for reconfiguration. Then, we look at system startup to give an intuition how the system is set up. Then, we describe the main protocol that is run for each type of reconfiguration. As mentioned above each start or crash of an LB server S triggers group changes for all groups associated with S. Upon the group change of the LB group, we run an LB coordination, upon the group change of a replication group,

we start an RM coordination. These coordination protocols make sure that all LB servers have up-to-date information about the state of the system and can work properly. After discussing the coordination protocols, we look at the crash and the recovery of LB servers, and how clients communicate with the system after such reconfiguration.

5.1 Terms, definitions and data structures

5.1.1 Initial configuration

A configuration file contains all information needed to set up an LB system. It contains a list of all LB servers and a list of all replication groups. For each replication group G, it contains the list of servers that should have an RM for G, and the server who is supposed to be the primary for G. The file also indicates the load-balancing algorithm that should be used. Currently, we assume this configuration file is static and does not change. It is created once before system initialization and used at system start-up and whenever a server restarts after a failure.

5.1.2 Failover types

If an LB server S fails, all RMs on S fail as well. There are three types of failover schemes that could possibly happen in the LB system.

1. *groupignore:* If the failed LB server S has only backup RMs, each failed RM is simply removed from the server list of the corresponding group. No further reconfiguration is needed.
2. *groupupdate:* If there is a primary RM for group G on the failed LB server S, and if there exists an LB server S' that has a backup RM for G and no primary RM for any other replication group, then this backup RM on S' will become

the new primary RM for G. Group G is simply updated with a new primary RM.

3. *groupmerge:* If there is a primary RM for group G on the failed LB server S, and on every LB server, on which there is a backup RM for G, there is a primary RM for another replication group, none of the backup RMs can become primary for G since we allow at most one primary RM on each LB server. This means G has to be merged with another group G'. AN LB server S' is chosen such that S' has the primary RM for G' and a backup for G. The LB on S' will then merge group G into G'. Group G will not be available after the merge.

5.1.3 Group timestamps

Group timestamp is the mechanism used in the LB system to distinguish clients of different replication groups. A group timestamp *group_gts* is a value pair (G : *gts*) consisting of the name of the replication group (G) and its startup timestamp (*gts*). We will see later in more detail when and how such timestamps are generated. Roughly, a group receives a new timestamp whenever it changes the primary RM. Since the load-balancing algorithm assures sticky client assignment, we want to be sure that the same client always goes to the primary RM of the replication group that is responsible for the client. The first time the client connects to the replication group G assigned by the primary LB, it does not have the group timestamp of G. But when it receives the first response, it obtains the group timestamp of G with this response. After that, the client will piggyback the group timestamp of G in each request. We will see later that the group timestamp plays a critical role in the LB

system to guarantee that client requests are taken care of correctly even during and after reconfiguration. Group timestamps are maintained within two data structures.

- *Vector newGTS:* stores the group timestamps of all the currently available groups. The available groups are the groups that have a running primary RM.
- *Hashtable oldGTS:* stores key/value pairs respecting two group timestamps. The group timestamp stored as the key is the group timestamp $(G : t1)$ of a group G at the time of failure of the primary RM of G. The group timestamp $(G' : t2)$ stored as the value is the group timetamp of the current group G' which takes care of the clients of G after the crash of the primary of G. $G = G'$ in case of *groupupdate*, but $G \neq G'$ in case of *groupmerge*. *oldGTS* is necessary for the LB system to correctly handle failover and make sure the client requests always go to the right group after failover. All the key/value pairs in *oldGTS* must be updated each time after a failover to make sure that each failed group is always paired to a current running group which takes care of its clients.

5.1.4 Status information

Each RM in the system keeps track about the status of its replication group. Most of this status information already exists in the Adapt SIB system. Additionally, each LB keeps track about the entire current configuration in the system.

Status information at RM

This is basically the same information as already maintained in the Adapt SIB system. Each RM r of a group G has a flag *isPrimaryRM* indicating whether it is primary or not. When discussing the Adapt SIB system, we had seen that each RM r also has a *serverlist* containing all available RMs. The first RM in *serverlist* is

the primary RM. This information, however, is not that crucial anymore in the LB system, because the LB maintains such serverlists.

Status information at LB

The LB l of LB server S keeps track of its servername S. It also has flag *isPrimaryLB* indicating whether it is the primary LB. For each RM r running on S, l maintains a *RMInfoObject* identifying the name of r's group G, whether r is primary of G, and the group timestamp in case r is primary. For instance, $(G1, true, (G1 : gts))$ indicates that this RM is a primary for group $G1$ with group timestamp $(G1 : gts)$. The LB keeps all *RMInfoObjects* in a *RMITable*. The LB also keeps the *newGTS* vector and the *oldGTS* table described above. Furthermore, it maintains two lists that provide information about the configurations of the currently existing replication groups. The *groupserverlist* stores information of all replication groups for which at least one RM is running in the system, including both replication groups with primary RM and those without primary RM. A group without a primary RM could happen during the initialization of the LB system or when a group G merges with another group G' due to a failure of its primary RM such that G is disabled. For instance, if the *groupserverlist* contains an entry $(G, (S1, S2))$, it indicates that group G is up and has currently available RM members on sites $S1$ and $S2$. *available_serverlist* is a subset of the *groupserverlist*. It stores information of all replication groups with a running primary RM. That means these replication groups are ready to handle client requests. For instance, an entry of *available_serverlist* can be $(G, (S1, S2))$. This indicates that G has currently two running RMs, on $S1$ and $S2$ respectively. As in the Adapt SIB system, the first in the list is the primary RM. That is, in our

example, $S1$ is the primary RM. The *available_serverlist* is used by the LB to assign new clients to replication groups via the load-balancing algorithm. The second part $(S1, S2)$ of the entry $(G, (S1, S2))$ for G in *available_serverlist* replaces the original *serverlist* in the Adapt SIB sytem. That is, this is the list sent to the CRM once the load-balancing algorithm has decided that the client should be served by group G. Both *groupserverlist* and *available_serverlist* are dynamic since they will change due to failure and recovery in the LB system. However, after a reconfiguration has been completed, the lists on all LBs are identical.

5.2 Initialization of the LB system

At initialization time of the entire system, one LB server after the other is started up. For each server S, first the LB of S is initialized. The LB initialization triggers the initialization of the RMs on this server. In our current implementation, the number of LB servers, the number of replication groups and the assignment of RMs to replication groups are fixed as configured in the system configuration file. Furthermore, we assume that all LB servers configured in the configuration file are started up during system initialization. It will be part of future work to dynamically adapt the configuration during runtime. The most critical thing here will be to add completely new servers and new groups to the system

5.2.1 LB initialization

At the initialization of LB l on LB server S, l first reads the configuration file, and does some variable initialization. In particular, it sets its servername to S, and it sets *isPrimaryLB* to false. It initializes all tables, vectors, and lists. Then, it creates the *RMIInfoObjects* for all RMs that are supposed to run on S according to

the configuration file, and then calls the RM initialization procedure for each of them. The RM initialization is discussed in the next section. After all RMs are initialized, each RM has joined its replication group and has its proper status information. This information is also reflected back in the *RMIInfoObjects* of the LB. l then joins the LB group via a group communication join request. A group change event will be delivered to l and all LBs that are already members of the LB group (the ones that started up before l). The view change event triggers a coordination among the members of the LB group in which they exchange enough information such that each LB has updated status information that reflects the current configuration. In particular, if S is the first server to be started up, i.e., l is the first in the LB group, then l will be the primary LB, otherwise it will be a backup LB. We discuss this configuration in detail later. After this configuration process has finished, the LB and RMs of the new LB server are ready to perform their tasks.

5.2.2 RM initialization

The RM initialization procedure is called by the LB of LB server S for each RM r of group G configured to be running on S. r first sets *isPrimaryRM* to false, and *isconfiguredPrimary* according to the configuration file to either true or false. Then it joins G by submitting a group join request. A group change event will be delivered to r and all RMs that are already members of the G (the ones that started up before r). The view change event triggers a coordination among the current members of G in which they exchange enough information such that each RM has up-to-date information about the state of the group. This coordination protocol is described in more detail later. In particular, if r is configured to be primary, *isPrimaryRM* will

actually be true after the coordination, otherwise r will be backup RM. That is, if an LB server S is configured to have a backup RM running for a group G, then this RM will be a backup after RM initialization even if the primary RM of G is not yet running. Hence, during initialization time, a replication group can have backup RMs running without a primary RM. In contrast and as discussed in the previous section, for LBs the first LB to start up will be the primary LB in order to guarantee that the LB group always has a primary LB. Note also that since RM initialization takes place before the LB joins the LB group, the RM coordination always takes place before the LB coordination.

5.3 Coordination

As we have seen above, at system initialization LBs and RMs join the respective groups. Each join triggers a view change event at all old members of the group and the new member. This view change event, in turn, triggers a coordination among all group members to exchange status information. This coordination is not only run at system initialization but always when a group configuration changes, i.e., also when an LB server fails or restarts. When an LB server fails, its LB is excluded from the LB group and its RMs are excluded from their replication groups. When it restarts, the LB and RMs again rejoin their groups. The coordination protocols for LB and replication groups are the heart of our reconfiguration.

When a view change event is delivered by the GCS at all members of the new configuration of a group (including a joining member, excluding a leaving member), each member of this new group configuration multicasts a message to all members of the group. The content of the message reflects what the member knows about its

configuration and the status of the system. After each member receives the messages from all members of the group, it has the correct status information of this group and coordination for this member has finished.

The following discussion makes the following assumptions. When an LB server S starts at system initialization or restarts after a crash, first all RMs of S join their replication groups triggering the coordination within these groups. Then, the LB of S joins the LB group triggering the coordination within the LB group. By assuming such ordering, we can be sure that the LB of S, when sending its status information to the other LBs during coordination, it has the correct information about all RMs and their replication groups running on S.

In contrast, when an S crashes, the group communication system automatically triggers view change events for the LB group and all replication groups that had members on S excluding these members from their groups. These view change events can occur concurrently, and no particular order of these events can be assumed.

5.3.1 The RM coordination protocol

The purpose of the RM coordination of group G after a view change event is to let each member of G have up-to-date information of the group configuration. In particular, we must determine who is primary RM in G. RM coordination already took place in the Adapt SIB protocol. We simply adjust this protocol here to exchange and generate also information in the regard to the LB system. In particular, the *RMIInfoObject* that belongs to the LB must be updated at the end of the RM coordination if it has changed. Note that the RMs do not need to take care of the *serverlist* anymore, since the CRMs now receive this information from the LBs.

Let V be the view change event indicating that an RM r' has joined/left the replication group G. If r' joins the group, V is the first view it receives (after startup or crash). If r' leaves the group it does not receive V. Each member r of V can tell whether a V was triggered by a join of a new member r' or the leaving of an old member r'. Figure 5–1 shows the coordination steps taken by RM r of V upon receiving view change event V.

1. In case r' leaving the group, do nothing
2. In case r' joining the group
   (a) multicast whether r is primary (identified by *isPrimaryRM* variable)
   (b) receive all messages from all other members of V
   (c) if someone is already primary
      i. if ($r == r'$) (r is the one that joined G)
         * update *RMIInfoObject* of group G of local LB to: (G, *false*, *NULL*)
      ii. perform RM recovery for r'
   (d) else (no primary yet)
      i. if ($r == r'$) and is configured to be primary (identified by *isconfiguredPrimary*)
         * set *isPrimary* to true
         * create a new group timestamp (G:*gts*)
         * update the *RMIInfoObject* of group G of local LB to: (G, *true*, (G:*gts*))
      ii. else if ($r == r'$) and is not configured to be primary
         * update the *RMIInfoObject* of group G of local LB to: (G, *false*, *NULL*)
      iii. else ($r <> r'$) do nothing

Figure 5–1: Coordination at RM r of server S upon receiving view change event V indicating join/leave of RM r' of server S'

When an RM leaves its replication group G, failover has to be performed if this RM was the primary RM. In this case, either *groupupdate* or *groupmerge* occurs. Hence, in the LB system, the replication group G itself can not decide whether and which backup will become the new primary RM. Instead, the primary LB will make

such a decision. Hence, RM failover is not triggered by the RM coordination but by the LB coordination as we will see later. In fact, RM coordination actually does nothing when an RM leaves its group (line 1).

In case an RM has joined the group, the RM members exchange messages to determine whether there is a primary RM (lines 2a-b). We can consider several cases. First, let's have a look what happens during initialization. Assume LB servers are started up in order $S1, S2, S3$, and $S2$ is configured to have the primary RM for G. At startup of $S1$, G will be a group without primary (line 2d.ii). After startup of $S2$, since G has not yet a primary and $S2$ is configured to be primary, the RM on $S2$ will set itself as primary and inform its LB (line 2d.i). We will later see that the LB during LB coordination will make G available for processing client requests. After startup of $S3$, G has already a primary and has possibly already executed client requests. Hence, the backup RM on $S3$ has to go through recovery (line 2c). Now assume, that after initialization of all three LB servers, $S2$ fails. Assume a *groupupdate* happens and $S3$ has the new primary RM of G. If now $S2$ rejoins, although $S2$ is configured to have the primary RM for G, there is already a primary at $S3$. Hence, the RM on $S2$ will be a backup RM and needs recovery (line 2c). If, however, a *groupmerge* occurred after the crash of $S2$, then G became disabled at the time of $S2$'s crash. When now $S2$ rejoins, its RM can again be the primary RM (line 2d.i). Note that in all cases 2c and 2d, only the joining site updates its *RMIInfoObject*. For an existing RM in G, the *RMIInfoObject* will not change. In fact, the only thing existing RMs might do is to help the new RM in recovery (line 2.c.ii).

5.3.2 The LB coordination protocol

The main purpose of the LB coordination is to choose the primary LB if there is no primary, and to let each running LB have the up-to-date information of the whole LB system, in particular the appropriate *groupserverlist* and *available_serverlist*. They are needed for load-balancing and to provide the CRMs with correct serverlists.

Let V be the view change event indicating that an LB has joined/left the LB group. If an LB joined the group, V is the first view it receives (after startup or crash). If an LB leaves the group it does not receive V. Each member l of V can tell whether V was triggered by a member leaving the LB group or a new member joining the LB group. Figure 5–2 shows the steps of LB l running on server S upon receiving view change event V.

The LB coordination triggers the recalculation of *groupserverlist* and *available_serverlist*. Before resetting, however, we temporarily store the old *available_serverlist* which we will need for failover (lines 1-2). Then, each LB, including the new joining LB if it is the case, multicasts a message indicating whether it is currently primary of the LB group and all information about its RMs, and then waits to receive all the messages from all members (lines 3-4). LB coordination has to make sure that there is always exactly one primary LB (line 5). For that the first left-most LB server listed in the configuration file that is also member of the new view V will have the primary LB. Hence, at system initialization, when the first LB server starts up, its LB will set itself to be primary LB (independently of where it is listed in the LB server list of the configuration file). After that, when the remaining servers start up, there is already a primary LB, and they will become backup LBs. After that, only if

1. Reset *groupserverlist*
2. Set *old_available* to *available_serverlist*, and then reset *available_serverlist*.
3. multicast whether l is primary (identified by *isPrimaryLB* variable) and all *RMIInfoObjects* (stored in *RMITable*) of l
4. receive all messages from all members of V.
5. if none of the LBs is primary LB
   (a) go through list of LB servers in configuration file from left to right
   (b) let S' be the first in this list such that l' running on S' member of V
   (c) if ($l == l'$) set *isPrimaryLB* to true
6. for each message received from LB l' running on server S' (including own one) and for each *RMIInfoObject* of group G included in message
   * if S' has primary RM for G
     (a) make a new entry for G in *groupserverlist* if there was not one yet or adjust the entry for G by making S' the first one in the server list *serverlist* of G in *groupserverlist*
     (b) create entry (*G, serverlist*) in *available_serverlist*
     (c) If l' joined V, take (*G:gts*) of the *RMIInfoObject* of G, and include in *newGTS*.
   * else
     (a) make a new entry for G in *groupserverlist* if there was not one yet or adjust the entry for G to include S'
     (b) if an entry for G exists in *available_serverlist*, append S' to the entry for G if not already included
7. In case of an LB leaving the group initiate LB failover

Figure 5–2: Coordination at LB l upon receiving view change event V indicating join/leave of an LB

the primary LB crashes and leaves the LB group, another LB will take over as new primary.

Apart of deciding on a primary LB, the information about the RMs is used to rebuild *groupserverlist* and *available_serverlist* (lines 6). In case of a join, we can be sure that RM coordination takes place before LB coordination, Hence, the new LB has already correct information about the status of its RMs, and therefore, all $RMIInfoObjects$ transmitted in the messages include the correct information, and the LBs update their lists appropriately (line 6 (a,b)). In case a new LB server joins

and it has a primary RM for a group, then this is a new enabled group, and hence, its group timestamp must be stored in *newGTS* (line 6c). In case of a crash, the LB leaving the group does not participate in the coordination, and hence, its server will not appear anymore in the *groupserverlist* and *available_serverlist* of the remaining LBs. The question is, whether the lists will contain the correct information about the remaining servers considering that in case of crash, there is no guarantee on the order of coordination, i.e., LB coordination can take place before RM coordination or vice versa. However, the order is not important. Recall RM coordination as described in the previous section. When an RM leaves its replication group, the remaining members of the group actually do not exchange any messages. As mentioned before, in contrast to the SIB algorithm, the RMs don't decide by themselves who becomes the new primary RM if the old crashed. Instead, the LBs will take care of this (line 7), and we have a closer look at this in Section 5.4.1.

5.4 Failover

If an LB server fails, its LB and RMs fail as well and leave their corresponding groups. This automatically triggers the coordination protocol in each of these groups. As discussed in Section 5.3.1, RM coordination actually does not do anything in this case. Instead, the failover protocol at the LBs is responsible to decide on the fate of the replication groups to which the failed LB server belonged. For the backup RMs that were running on the failed LB server nothing has to be done. The only critical case is for the replication group G for which the failed LB server had the primary RM. Note that there is at most one such group G.

5.4.1 LB failover

LB failover consists of several steps. First, the primary LB decides what kind of failover has to be performed (*groupignore*, *groupmerge*, *groupupdate*) and informs the other LBs accordingly. Upon receiving the decision, failover is completed if nothing has to be done. This is the case if the failed LB server only had backup RMs. Otherwise, one LB is responsible for performing the *groupmerge/groupupdate*. Once it is completed, it informs all other LBs that failover has completed, and all LBs adjust their data structures accordingly. Figure 5–3 shows the steps of LB l running on server S performed for failover.

In step 1, the primary LB decides on what has to be done. By comparing *old_available* and *available_serverlist*, the primary LB can decide whether there was a primary RM running on the crashed server. If not, nothing has to be done (line 1a). If a primary RM for group G was running (line 1b.i), the primary LB can look in *groupserverlist* to determine backup RMs for group G (line 2b.ii). For each of these servers S', the primary LB checks in *available_serverlist* whether this server has a primary RM for another group G'. This is the case if there is an entry $(G', (S', ...))$ in *available_serverlist* where S' is the first in the serverlist. If there is a server S' with a backup for G and no primary RM for another group, then a *groupupdate* is possible. The primary LB sends the decision for *groupupdate* to all, indicating that S' should run the new primary RM for G (line 1b.iii). If all servers with a backup RM for G have also a primary RM running, then the primary LB sends the decision for *groupmerge* to all, choosing any of the servers S' with a backup RM for G to

1. if pimary LB then (decide on what has to be done)
   (a) if *old_available* has entries for the same groups as *available_serverlist*,
   then multicast decision = (*groupignore*) (failed LB server had only backup RMs)
   (b) else
      i. let G be the group failed LB server had primary RM (group appears in *old_available* but not *available_serverlist*)
      ii. let *serverlist* be the list of servers associated with the entry for G in *groupserverlist*
      iii. for the first server S' in *serverlist* that has no primary RM for another replication group (determined by looking at *available_serverlist*), multicast decision = (*groupupdate, G, S'*) (indicating that S' should failover to have the new primary for G).
      iv. if there is no such server (all servers with backups also run a primary RM), pick any server S' in *serverlist*, multicast decision = (*groupmerge, G, S'*) (indicating that S' should merge the group for which it has the primary RM with G).
2. receive decision
3. if decision == (*groupignore*), return
4. if decision == (*groupupdate, G, S'*) and local server $S == S'$
   (a) call *RM failover* of the backup RM of group G on local server S
   (b) create a new group timestamp (*G:gts*)
   (c) update the *RMIInfoObject* of group G to: (*G, true,* (*G:gts*))
   (d) multicast success = (*groupupdate, G, S',* (*G:gts*))
5. if decision == (*groupmerge, G, S'*) and local server $S == S'$
   (a) let G' be the group for which S has primary RM
   (b) call *RM merge* of the primary RM of group G' on local server S indicating G as group to be merged
   (c) multicast success = (*groupmerge, G, S' ,G'*)
6. receive success message
7. if success == (*groupupdate, G, S',* (*G:gts*))
   (a) update *available_serverlist* to have entry for G (take entry for G from *groupserverlist* but set S' as first server in list)
   (b) let (*G:oldgts*) be the group timestamp of G in *newGTS*.
   (c) set (*G:gts*) as new group timestamp of G in *newGTS*.
   (d) include pair (*G:oldgts*)/(*G:gts*) as pair in *oldGTS*, and for each pair (*g:x*)/(*G:oldgts*) in *oldGTS*, replace with (*g:x*)/(*G:gts*).
8. if success == (*groupmerge, G, S' ,G'*)
   (a) if local server S has backup RM for G, call *RM reset*
   (b) let (*G:oldgts*) be the group timestamp of G in *newGTS*.
   (c) let (*G' :gts*) be the group timestamp of G in *newGTS*.
   (d) remove (*G:oldgts*) from *newGTS*.
   (e) include pair (*G:oldgts*)/(*G' gts*) as pair in *oldGTS*, and for each pair (*g:x*)/(*G:oldgts*) in *oldGTS*, replace with (*g:x*)/(*G' :gts*).

Figure 5–3: LB failover steps at LB l of server S

perform a merge of G with the group G' for which S' has the primary RM (line 1b.iv).

Each LB receives the decision message (line 2). Failover is completed if nothing has to be done (line 3). In case of *groupupdate* (line 4), the LB on the responsible server S' initiates RM failover so that its backup RM for G becomes the new primary for G taking over all current clients of G. The RM failover procedure is basically the same as in the Adapt SIB system. Once this failover is completed, the LB of S' generates a new group timestamp, udpates its $RMIInfoObject$ data structure and multicasts a success message. In case of *groupmerge* (line 5), the LB on the responsible server S' determines for which group G' it has the primary RM. It can determine G' by looking at its $RMIInfoObjects$. Then, the LB calls a special merge procedure of the primary RM of G' running on S'. We will see later how this merge procedure will merge the backup information of G with the primary information of G'. G' will take over all current clients of G. After successful merge, the LB multicasts a success message.

Each LB receives the success message unless nothing needed to be done (line 6). If a *groupupdate* took place (line 7), G remains an available group with a primary RM. Hence, the *available_serverlist* must be updated. Since G changed its group timestamp, the data structures $newGTS$ and $oldGTS$ must be updated accordingly. If a *groupmerge* took place (line 8), G is no more available, i.e., it has no primary RM anymore but another group G' will take over the clients of G. Hence, any LB server that has a backup RM for G resets this backup (line 8a). This is necessary because when G becomes again an available group later on, it will not serve its old

clients anymore but only accept new clients. Furthermore, since G is now disabled, its timestamp is removed from *newGTS*. G' remains its current timestamp. Additionally, we capture in *oldGTS* that now G' is responsible for the current clients of G.

After the failover procedure terminates at an LB l, all its data structures capture the latest information of all replication groups, independently of whether l is primary LB or not. Hence, if the primary LB fails, any other LB can immediately take over as new primary LB.

5.4.2 Failover tasks at RMs

We have seen above that the LB makes three different types of requests to a local RM. In case a backup RM of a group G should become primary RM, the LB calls the standard failover procedure, that already existed in the SIB algorithm. It returns when the new primary RM is ready to receive client requests. In case a replication group is disabled, the remaining backup RMs have to be reset. This is quite simple. They simply discard all the information about EJBs, clients and responses.

The steps for a group merge are more complicated. In case of a group merge on a server S' that has a primary RM for group G' and a backup RM for group G, the primary *RM* of G' must take the backup information of G, perform failover for G according to the SIB algorithm, and then merge the data structures maintained by the SIB algorithm for G' with those of G. Most of the merge is straightforward, since G and G' so far served different sets of clients. Hence, they have disjoint sets of SFBS and request/response pairs. However, clients of both groups might have accessed the same EBs. This is possible if the accesses in both groups were read accesses. In this case, the merge has to take care that there are not two instances

of the same EB. The primary RM of G' must also multicast the backup information of G to all group members of G' such that all the backup RMs of G' have all the replication information of group G. This is needed because not only the primary of G' must be able to handle requests from the clients of G but also the backups in G' must be able to receive now replication information about these clients during normal processing. The merge procedure returns when the primary RM of G' has completed the merge, is ready to serve the clients of both G and G', and the backup RMs of G' have received the backup information for G.

5.5 Recovery

Compared to failover, the recovery protocol is quite simple and completely covered by the coordination protocol. The join of a recovered LB server S triggers an RM coordination in each affected replication group and an LB coordination in the LB group. This coordination protocol takes complete care of recovery. Assume there is an RM r on S that is a configured primary RM for group G. After the RM coordination, r knows whether G currently has a primary RM or not. If yes, then it becomes a backup for G. However, if G currently is disabled and has no primary RM, which happens after a *groupmerge*, r becomes the new primary for G. Thus, G is available to the load-balancer and back to service for new clients.

5.6 Client requests after LB server crash

We have already discussed in the previous chapter, how clients are handled if no reconfiguration takes place. A main change compared to the Adapt SIB system is that a client now always communicates via the LB of a server and not with the RM directly. A client c connects to the system via the *initialContext* method. This

method accesses a file containing the LB serverlist. It contacts the first in this list as being the primary LB. This primary LB assigns a group G to c according to the load-balancing strategy and provides the *initialContext* with the current server list of group G (according to *available_serverlist*). When c now contacts the server S with the primary RM of G (the first in the list), it will transparently receive the CRM which will then intercept each client request according to the Adapt SIB system. In our LB system, the CRM is an LB stub (in contrast to the Adapt SIB system, where it was a stub of the RM). The CRM maintains a *request_gts* object which keeps track of the group timestamp information of the current group that the client is connected to. When the CRM intercepts a client request, the CRM piggybacks *request_gts* on this request and sends this request to the LB l of server S. If it is the first time the client makes a request, *request_gts* has a null value. If the CRM receives a failure exception to the request, it resends the request to LB l' of server S' which is the next in the server list of group G until it receives a positive response. A positive response might piggyback a group timestamp *group_gts*. If this is the case, the CRM sets *request_gts* to *group_gts*.

When an LB l on server S receives a request from a client c with *request_gts*, there are two possibilities.

Case 1) S has a primary RM *PRM* of group G with group timestamp *group_gts*. We consider several cases.

- If *request_gts* is null, that means this is the first time the client makes a call to the server S, l will pass this request to *PRM* (primary of G). It is possible that this client was originally assigned to group G' on LB server S', but S' failed

and S is the next available server on the server list of G'. Hence, when the CRM received a failure exception for S' it resent the request to S. However, since this is the first request of the client, the client will simply be assigned to G instead of G'. When the call returns, l piggybacks *group_gts* so that the CRM stores it in *request_gts* and piggybacks it on its next request.

- If *request_gts* is not null, l compares the *request_gts* to *group_gts*. If the two match perfectly, l will pass the request to *PRM* (primary of G), and return the response to the CRM. Otherwise, l checks its *oldGTS* table to see if there is a group timestamp which is paired with *request_gts*. This information might not be available if the LB system is currently undergoing a reconfiguration. In this case, we wait and check the *oldGTS* once the reconfiguration is finished. Let *new_gts* be the group timestamp which is paired with *request_gts*. This means the client with *request_gts* now should be taken care of by the group with *new_gts*. If *new_gts* matches *group_gts* of group G for which S has the primary RM, l passes the client request to *PRM* and returns the response to the CRM piggybacking *group_gts*. If *new_gts* does not match *group_gts*, then G is not the group that should take care of this client. l returns an exception to the CRM. The CRM will try the next server in the server list until it finds the right one. A match between *group_gts* and *new_gts* occurs in the following scenario. Let *request_gts* belong to group G' on server S'. The CRM had originally sent the request to server S'. However, S' failed and the CRM received a failure exception. At the crash of S' the LB system either performed a *groupupdate* or *groupmerge*. In any of the two cases, a server S'' with a backup RM of

G' took over the clients of G' and the *request_gts* was included in $oldGTS$ paired with the correct new group timestamp. When the CRM received the failure exception from S' it contacted the next server in server list of G'. If this is S and S happens to be the server S'' that took over the clients of G', then *new_gts* matches with *group_gts*, otherwise it does not. It is also very important to keep the paired information of $oldGTS$ up-to-date. For example, we have a pair $(G1, TS1)/(G2, TS2)$ in $oldGTS$, and now $G2$ fails and was taken over by $G3$, we then have a new pair of $(G2, TS2)/(G3, TS3)$ in $oldGTS$. We want to update the old pair of $(G1, TS1)/(G2, TS2)$ to $(G1, TS1)/(G3, TS3)$ such that we can find out that $G3$ is taking care of clients of old $G1$ by only one search in the oldGTS (see Figure 5–3 line 7d and 8e).

Case 2) S has no primary RM on it. In this case, if reconfiguration is currently ongoing, we wait until reconfiguration has finished. If S still has no primary RM, l returns an exception to the CRM. Otherwise, the actions under **Case 1** are performed.

While failover requires to redirect clients of a failed RM, the recovery of an LB server does not really affect the client request processing for the existing clients of the LB system. But if an RM on this recovered LB becomes a new primary of a disabled group and makes this group available again to the load-balancer, there will be one more group to handle new incoming clients.

5.7 Case study

In order to better understand how client requests are processed while and after reconfigurations, we give some typical examples of request processing within an LB system.

Case 1): *groupignore.* We have a system of three LB servers *S1*, *S2*, *S3* and the primary LB (PLB) is on *S1*. There are two replication groups: *G1*={*S1, S2, S3*}, where *S1* has the primary RM and the group timestamp is $(G1 : ts11)$, and *G2*={*S2, S1, S3*}, where *S2* has the primary RM and the group timestamp is $(G2 : ts21)$ (see Figure 5–4). There are only two backup RMs on *S3*. The server *S3* first fails and later recovers..

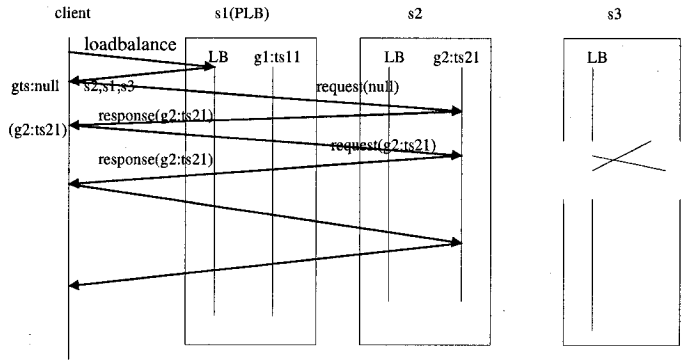


Figure 5–4: groupignore

In Figure 5–4, *S1* is the primary LB, and *S2* and *S3* are only backup LBs. *S1* has the primary RM of group *G1*, *S2* has the primary RM of group *G2*, and *S3* has only backup RMs. When a client first calls the LB system, the load-balancing algorithm

is triggered. Assume the client is assigned to group *G2* by the load-balancer. The client gets back the group server list *G2*= {*S2, S1, S3*}. At that time, *request_gts* at the CRM is null. The client sends its first request to *S2* and the CRM piggybacks *request_gts* with value null. Once the CRM gets back the response with the current group timestamp ($G2 : ts21$), the CRM sets its *request_gts* to ($G2 : ts21$). After that, the CRM always sends requests with *request_gts* as ($G2 : ts21$). Now assume that *S3* fails. Since there were only backup RMs on *S3*, no client request is affected. Later, *S3* recovers. No client request will be affected neither.

Case 2): *groupupdate*. The system setup is the same as in **Case 1**. We want to show how the client request processing is affected by the failover and recovery if *S2* fails and recovers later (see Figure 5–5). Note that *S2* has primary RM of *G2*. After the failure of *S2*, the backup RM of *G2* on *S3* becomes the new primary RM of *G2* and a new group timestamp ($G2 : ts22$) is assigned. When server *S2* recovers, the configured primary RM of *G2* on *S2* becomes a new backup since there is already a primary RM for *G2* on *S3*.

In Figure 5–5, a client connects to the system, and is assigned to *G2*={*S2,S1,S3*} by the load-balancer. The client sends its first request with *request_gts* as null to *S2*. Since the *request_gts* is null, the request is processed by the *RM2* on *S2*. Then a response piggybacking current group timestamp ($G2 : ts21$) is sent back to the client, and the CRM of this client sets its *request_gts* object to ($G2 : ts21$). Now assume that $S2$ crashes and failover starts. At the same time, the client sends its second request with *request_gts* as ($G2 : ts21$) to *S2*, but because *S2* has crashed, the CRM of the client gets back an exception. The CRM then tries the same request

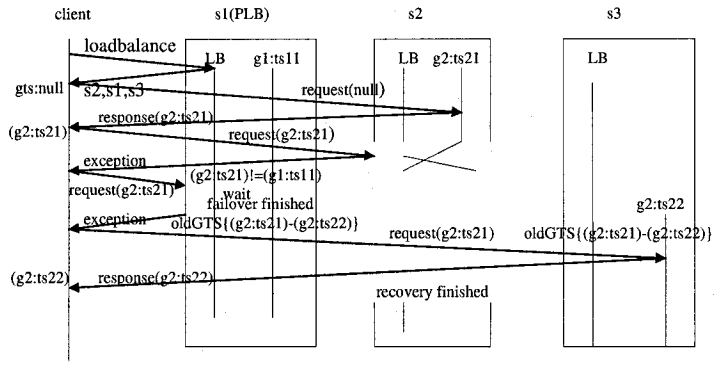


Figure 5–5: groupupdate

on the next server on its server list, which is *S1*. But since *request_gts* is $(G2 : ts21)$, and the *group_gts* on *S1* is $(G1 : ts11)$, they do not match. Since failover is not yet completed, the request has to wait. Once the failover is completed, $S3$ has the new primary for *G2* with group timestamp $(G2 : ts22)$. Now, the LB on *S1* checks in *oldGTS* for a match for $(G2 : ts21)$, and finds $(G2 : ts22)$. Since $(G2 : ts22)$ does not match $(G1 : ts11)$ of the primary RM on $S1$, an exception is sent back to the CRM again. Now the CRM sends the request with *request_gts* as $(G2 : ts21)$ to *S3*. The *request_gts* does not match the group timestamp $(G2 : ts22)$ of the primary RM running on *S3*. But after checking *oldGTS*, the LB on *S3* finds the match with $(G2 : ts22)$ and knows that its local primary RM for $G2$ is responsible for this client and forwards it to the RM. When returning the response to the CRM it piggybacks the new group timestamp $(G2 : ts22)$. When the CRM receives the response, it will set its *request_gts* object to $(G2 : ts22)$. Everything is back to normal again.

Now even if *S2* recovers, since there is already a primary RM for *G2*, the configured primary of *G2* on *S2* has to become a new backup of *G2*. Thus, recovery has no effect on request processing.

Case 3) *groupmerge* Assume a system with three LB servers, *S1*, *S2*, *S3*, and three replication groups. *G1*={*S1, S2, S3*}, where *S1* has the primary RM, and the group timestamp is $(G1 : ts11)$, *G2*={*S2, S1, S3*}, where *S2* has the primary RM and the group timestamp is $(G2 : ts21)$, and *G3*={*S3, S2, S1*}, where *S3* has the primary RM and the group timestamp is $(G3 : ts31)$. No server has only backup RMs. If server *S3* now fails, the primary LB has to make a *groupmerge* reconfiguration decision. Assume it chooses to merge *G3* with *G2*. After the failover, *G3* is not available anymore, and *G2* takes care of all the existing clients of *G3*. *G2* keeps its timestamp $(G2 : t21)$ and a pair $(G3 : t31)/(G2 : t21)$ is included in *oldGTS*. Once *S3* recovers, because there are only two backup RMs of *G3* on *S1* and *S2*, the configured primary RM of *G3* on *S3* becomes the new primary RM, and *G3* becomes a brand new *G3* with *(G3:ts32)*.

In Figure 5–6, we show two clients of the LB system. *client1* is an existing client of group *G3* with *request_gts* of its CRM set to $(G3 : ts31)$, and *client2* is a new client of the LB system. Assume it is also assigned to group *G3*. After the first request, the CRM of *client2* has *request_gts* set to $(G3 : ts31)$. Now $S3$ fails. When *client2* now sends its second request to $S3$, its CRM receives a failure exception. The CRM then sends the request again to the next server *S2*. Since the *request_gts* $(G3 : ts31)$ does not match the group timestamp $(G2 : ts21)$ of the primary RM of $S2$ the request has to wait until failover finishes. Once failover has

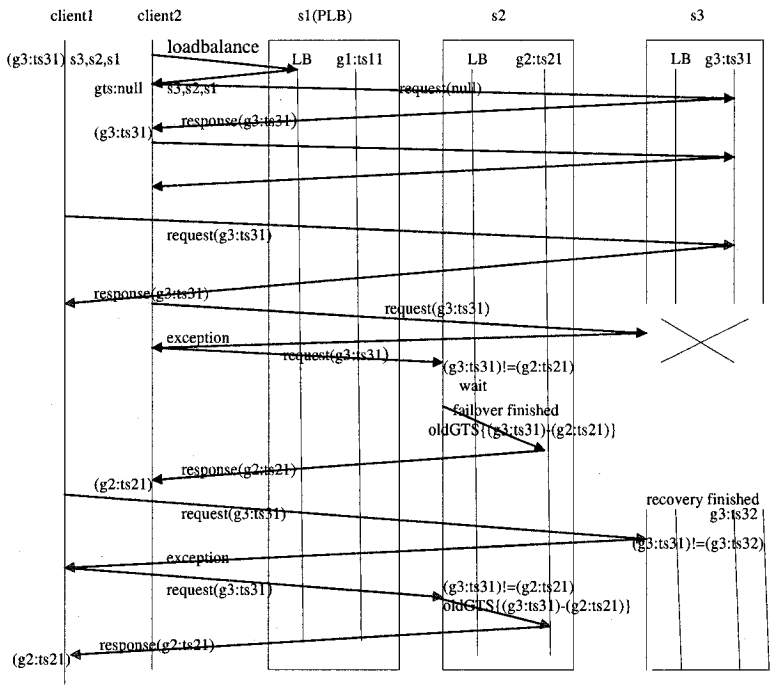


Figure 5–6: groupmerge

completed, group *G3* has merged with *G2*. The LB of *S2* checks in *oldGTS* and finds a match of $(G3 : ts31)$ with $(G2 : ts21)$ which is the current group timestamp of $G2$ for which $S2$ has the primary RM. Hence, the LB forwards the request to this primary RM and returns the response back to the CRM of *client2* piggybacking group timestamp $(G2 : ts21)$. The CRM will set its *request_gts* accordingly. Assume now that *S3* recovers and installs a new primary of *G3* with *(G3:ts32)*. $G3$ is now again available to the load-balancer. Now let us look at *client1*. *client1* did not send any request before *S3* is fully recovered. Suppose now *client1* sends a request

piggybacking *request_gts* ($G3 : ts31$) to *S3*. Since the *request_gts* does not match the current group timestamp ($G3 : ts32$) of $G3$, $S3$ checks *oldGTS* and finds a match with ($G2 : ts21$). *S3* realizes that *client1* has been taken over by group $G2$. Hence, it sends an exception back to the CRM of *client1*. The CRM then sends the same request to *S2*. The LB on *S2* also checks in *oldGTS*, finds the match for its local group, takes care of request execution, and sends the response back to the CRM piggybacking ($G2 : ts21$). The CRM will update its *request_gts* accordingly. That is, all clients $G3$ had at the time of the crash are taken over by group $G2$, and the special case where a client does not send requests between the crash and the recovery of a server is correctly taken care of.

CHAPTER 6
Experiments and Result Analysis

To evaluate the performance and the functionality of the LB system, we have conducted two sets of experiments. The first set of experiments compares the performance of the original JBoss, the Adapt SIB system and the LB system during normal processing (without failure and recovery). The purpose of these tests is to see how load-balancing can improve the performance in case that the application server is the bottleneck of the system. Another set of experiments is to test the behavior of the LB system during failure and recovery. The failure cases are *groupignore* (a backup fails), *groupupdate* (the primary fails, and a backup takes over as primary), and *groupmerge* (the primary fails and the group merges with another group). We would like to show the effect of each of these reconfigurations on the LB system. The load-balancing algorithm used in all these tests is *Round Robin.*

6.1 Hardware and software used in the experiments

1. Hardware We used four Linux computers with the names *cs8, cs9, cs10, cs11* (each has 3.4GHz Pentium 4 CPU with 1GB RAM). Three of them are used as LB servers, and one of them is used as a client simulator. They are all located in the same local network with a fast Ethernet connection.

2. Software We compared three different configurations: the original JBoss server, the Adapt SIB replication system, and the LB system. Furthermore, we had a client simulation program, which simulated the client access to the server. The

client simulation program had several parameters to be set such that it could simulate different scenarios. We can set the number of clients to run simultaneously, the total running time for each client, the number of transactions per second submitted by each client, and the application that the clients access.

As for applications, we used a set of three simple EJB applications with only one stateful session bean (SFSB) per client. Database access was always through the stateful session bean. The applications differ in how heavy the processing is required within the server and the database.

1. No database access: In this case, the SFSBs do some heavy computation on behalf of client requests leading to high CPU load on the application server. There is no database access at all. In this case, the application server may become the bottleneck of the system.

2. Only database access: The only task of the SFSB is to access the database and return the result to the client. The SFSB itself does not do any other processing. In this case, the database server could become the bottleneck of the system.

3. Database access plus SFSB processing: The SFSBs not only conduct database access but also do some processing after the database access. In this case, either the application server or the database server could become the bottleneck of the system.

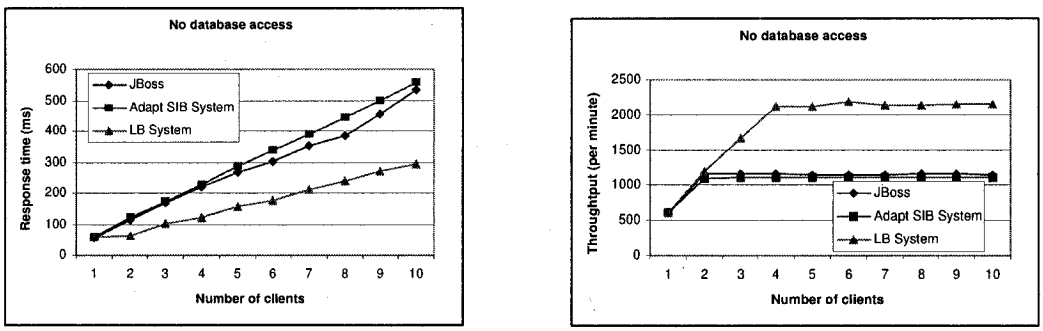
6.2 Performance tests during normal processing

6.2.1 System configuration

In all cases, the client simulation program runs on *cs8*. When running the original JBoss system without replication, only one machine is used for JBoss (*cs9*).

For the Adapt SIB system, three replicas are running on *cs9, cs10, cs11* respectively, with *cs9* being the primary. The LB system has two groups, and each group has three members: *G1*= {*cs9, cs10, cs11*} with primary running on *cs9*, *G2*= {*cs10, cs9, cs11*} with primary running on *cs10*. The primary LB will be decided during the LB system initialization (the one that is first started up).

6.2.2 No database access



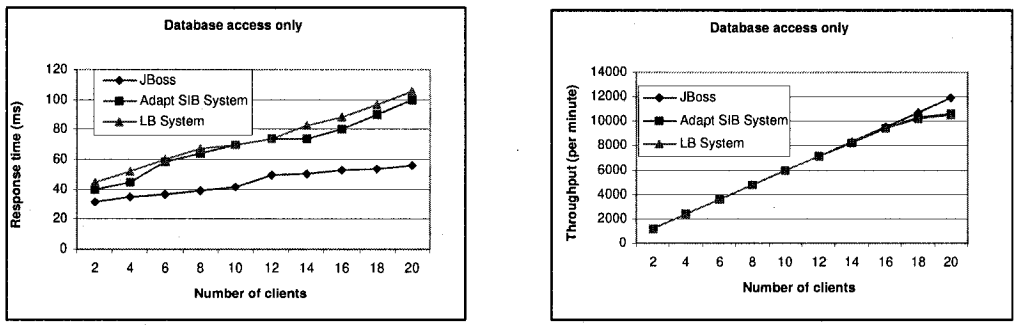
(a)Response time vs Number of clients (b)Throughput vs Number of clients

Figure 6–1: No database access

Figure 6–1 (*a*) shows the response time with increasing number of clients for all three systems, in which each client submits 10 transactions per second. All of them have the same pattern, the response time increases almost linearly with the increase of the number of clients. Compared to the original JBoss system, the Adapt SIB replication system has slightly longer response time, which is caused by the replication overhead. The LB system has the shortest response time among all three systems. The LB system has more overhead than the Adapt SIB system since the LB system has more groups and accordingly more group communication. But because the LB system actually has two replication groups, and each group has its own

primary RM located on a different machine, each primary RM serves an average of half of the clients. Hence, each machine is less loaded leading to the overall lower response times than the Adapt SIB system or JBoss. For instance, at Figure 6–1 (a), LB has response times nearly 50 percent lower than both Adapt SIB and original JBoss. We can also see this clearly from Figure 6–1 (b) which shows the maximam achievable throughput depending on the number of clients. The original JBoss system and the Adapt SIB system both saturate when the client number reaches 2. But because the LB system has two replication groups, i.e., two LB servers handle client requests, the LB system becomes saturated when the client number reaches 4. After the saturation point, the throughput of each system stops increasing.

6.2.3 Database access only



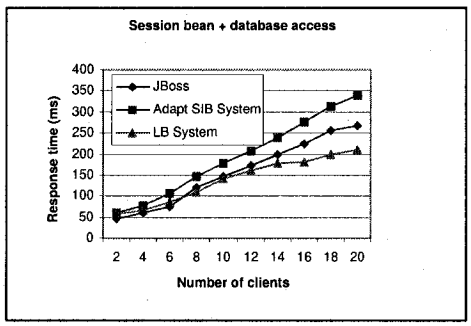
(a)Response time vs Number of clients (b)Throughput vs Number of clients

Figure 6–2: Database access only

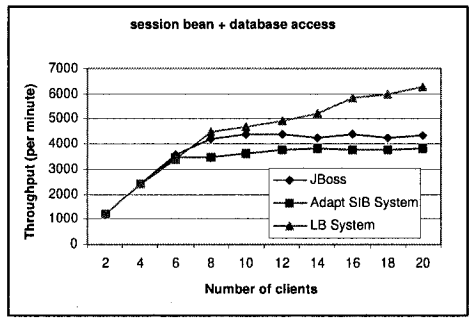
In this test, each client also submits 10 transactions per second. Figure 6–2 shows response time and throughput with increasing number of clients where the application mainly accesses the database. In this test, the response time is mainly the database access time plus the time used to insert a marker into the database for

the SIB system and the LB system. The application server has the least impact on the response time, which also means the application server can never become the bottleneck of the system. Because in this experiment, the application server never becomes the bottleneck of the system, adding more primary servers cannot improve the system performance. Instead, it increases the overhead of the system. The LB system does not have any advantage in this situation and it has the worst response time among all three systems due to the increased overhead. However, Figure 6–2 (*a*) and Figure 6–2 (*b*) shows that LB performs nearly as good as SIB despite the higher communication overhead. In general, however, inserting an additional marker, both needed for the SIB system and the LB system puts even more burden on the DB leading to considerable worse response time and slightly worse max throughput than a non-replicated system.

6.2.4 Database access plus SFSB processing



(a)Response time vs Number of clients



(b)Throughput vs Number of clients

Figure 6–3: Database access plus SFSB processing

Again in this test, each client submits 10 transactions per second. Figure 6–3 shows response time and throughput with increasing number of clients where the

application accesses the database and also does some work in the application server. In this test, the response time consists of two parts, half of the time for database access, including the time needed to insert a marker into the database for the SIB system and the LB system, another half of the time for the processing within SFSBs. The application server can not do anything to improve the database response time, but it can do something to improve the response time of SFSBs. The LB system has more overhead indeed, but because the LB system improves its response time by using load-balancing to distribute the load into two groups, the overall response time is still the lowest among the three systems (see Figure 6–3 (*a*)). Also, the LB system can handle more throughput than both the original JBoss system and the SIB system (see Figure 6–3 (*b*)).

From the above experiments, we conclude that if the application server is the bottleneck of the whole system, we can improve system performance by using more primary servers and balance the load among them. But if the application server is not the bottleneck, the load-balancing mechanism can not improve the performance of the system. Instead, it adds more overhead.

6.3 Performance tests during reconfiguration

In this section, we show how the system behaves during reconfiguration (including *groupignore*, *groupupdate*, and *groupmerge*). *groupignore* means the failed server S only had backup RMs on it. In this case, the reconfiguration does actually nothing. *groupupdate* means that the failed server S had a primary RM for group G, and there is another LB server S' in the LB system which has no primary RM but at least a backup RM of the failed group G. In this case, the reconfiguration is to

update this backup RM to become the new primary of G. *groupmerge* means that the failed server S had a primary RM for G, and each LB server in the LB system which has a backup RM of G, also has a primary RM for another replication group. The reconfiguration is to merge G into another available group G'. In these experiments we first run the LB system for some time, then we force a server replica to crash. Finally we rejoin this replica to the system. We measure the response time every 20 milliseconds at the server side, and see how reconfiguration affects the response time. In all the following three tests, we let each client submit 10 transactions per second.

6.3.1 System configuration for both *groupignore* and *groupupdate*

In order to test *groupignore* and *groupupdate*, we start with a system configuration with *G1*={*cs9, cs10, cs11*} with *cs9* as the primary RM, *G2*={*cs10, cs9, cs11*} with *cs10* as the primary RM. It is obvious that there are only two backup RMs on server *cs11* (one for each group). The client simulation program runs again on machine *cs8*. The application used is the one without database access mentioned in section 6.1.

6.3.2 Experiment design for both *groupignore* and *groupupdate*

The experiment runs as follows. We first start up all three servers. Then we start four clients, each runs 200 seconds. We crash *cs11* and restart it later for the *groupignore*, and we crash *cs10* and restart it later for the *groupupdate*. Furthermore, we start four new clients before the four existing clients finish, which means there is a certain time period where there are eight clients in total in the system.

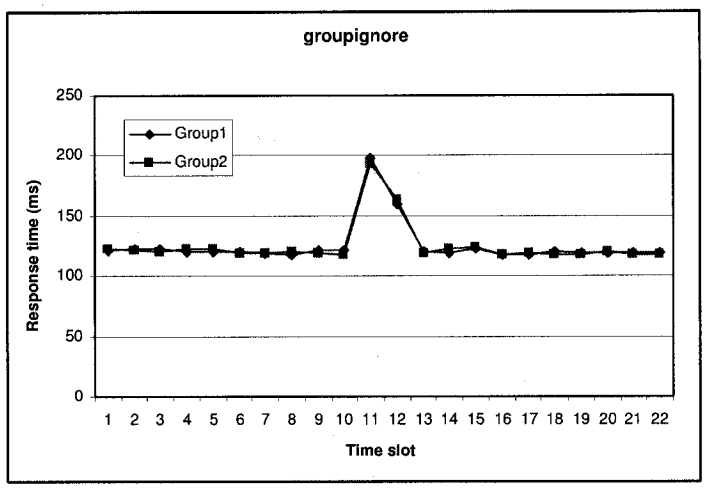


Figure 6–4: groupignore

6.3.3 *groupignore*

Figure 6–4 shows the response time measured at the server side. At the beginning, the system has only four clients, two for each replication group. *cs11* fails at time slot 7 and then recovers at time slot 8. Because there were only backup RMs on server *cs11*, the crash of cs11 does not affect any replication group. Hence, response times do not change during the reconfiguration. The recovery of server cs11 does not affect the response time neither. There is only backup RMs on it and the handling of the recovery messages does not affect the response time of the groups. At time slot 10, the second four clients start running in the system, which means there are in total eight clients in the system, four for each group. The response time increases due to the increase of the number of clients distributed on *cs9* and *cs10*. At time slot 13, the first four clients finish running, and there are only four clients left in the system, the response time decreases again due to the decrease of the number of clients.

Because the reconfiguration does not really do anything special, the reconfiguration here has no impact at all on the response time of any group.

6.3.4 *groupupdate*

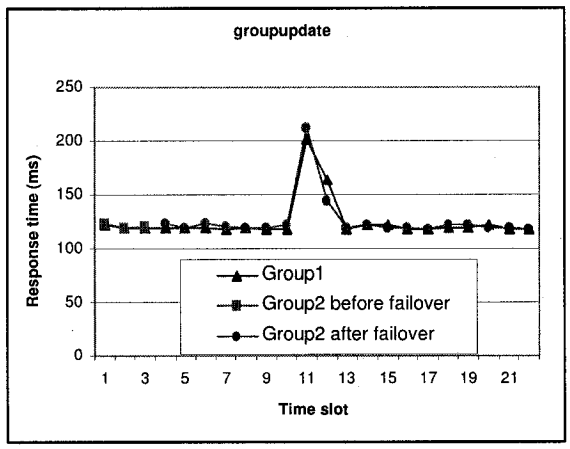


Figure 6–5: groupupdate

Figure 6–5 presents the response time measured at the server side when server *cs10* fails and recovers later. At the beginning, the system has only four clients, two for each group. At time slot 3, server *cs10* fails. Because the primary RM of group *G2* was on server *cs10*, *G2* has to find a new primary RM for its group. Since there are only backup RMs on server *cs11*, and one of them is a backup RM of *G2*, the reconfiguration updates this backup RM to become the new primary RM of *G2*. After the reconfiguration, *G2* continues to work as before except now the primary is on *cs11*. The response time of *G2* does not change much compared to group *G1* since failover is fast. At time slot 6, server *cs10* recovers, but since there is already a primary RM in *G2*, server *cs10* only can have two backup RMs, one for each group. The recovery of server *cs10* does not affect the performance of any groups since it

has only backup RMs. At time slot 10, the second four clients started, which means that there are eight clients in total in the system, four for each replication group. The response time of both group increases with the increased number of clients in the system. At time slot 13, the first four clients finish running, the response time decreases again due to the decrease number of clients in the system (from eight to four).

6.3.5 System configuration for *groupmerge*

In order to test *groupmerge*, we start with a system configuration with *G1*={*cs9, cs10, cs11*} with *cs9* as the primary RM, *G2*={*cs10, cs9, cs11*} with *cs10* as the primary RM, and *G3*={*cs11, cs9, cs10*} with *cs11* as the primary RM. It is obvious that each server has a primary RM. The client simulation program runs again on machine *cs8*.

6.3.6 Experiment design for *groupmerge*

The experiment runs as follows. We first start up all three servers. Then we start the first six clients, each runs 200 seconds. We crash *cs11* while the first six clients are still running. We then start up the second six clients before the first six clients finish running. This means during a certain time period, there are twelve clients in total in the system. After the first six client have finished running, we restart *cs11*. And then we start the last six clients before the second six clients finish running. This means there is another period of time where there are twelve clients in total in the system.

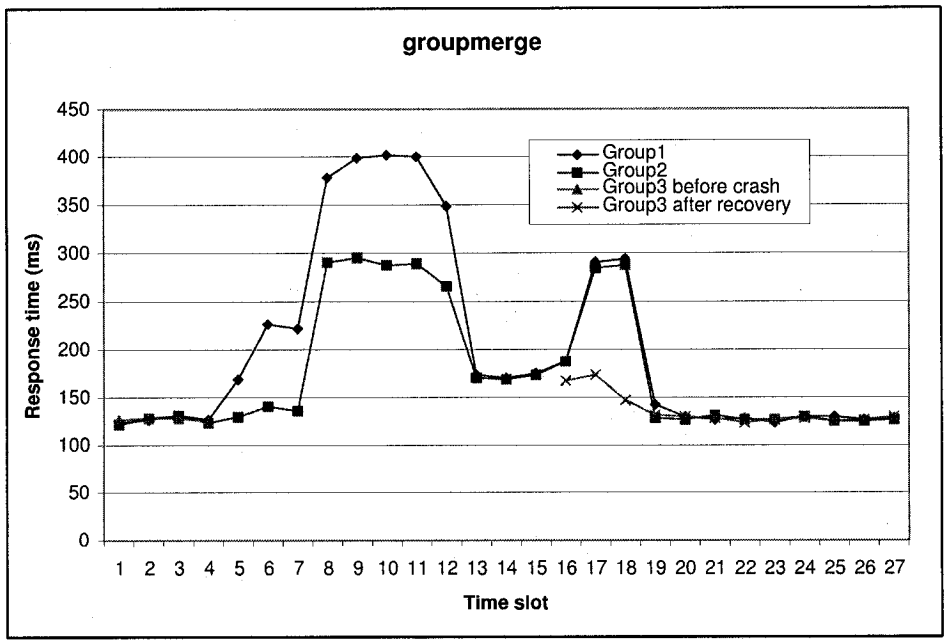


Figure 6–6: groupmerge

6.3.7 *groupmerge*

Figure 6–6 shows the response time during the reconfiguration measured at the server side. The first six clients start running in the system at time slot 1, each group has two clients, and they have almost the same response time. At time slot 4, server *cs11* fails, which means group *G3* has lost its primary RM. Since both *cs9* and *cs11* each has a primary RM, it is not possible in this case to do a *groupupdate* reconfiguration, but a *groupmerge* reconfiguration is performed. The LB system decided that group *G3* should merge with group *G1*, which means *G1* should take over the clients of *G3*. After the reconfiguration, *G3* is disabled, and is not available anymore to the clients of the system. We see clearly from Figure 6–6 that the

response time of *G1* increases after the time slot 4 because the two clients of group *G3* are now running in *G1*. In other words, there are four clients in total in *G1* while *G2* still has two clients. At time slot 7, the second six clients start before the first six clients finished running. Since *G3* is not available, there are only two groups available to the new clients. *G1* and *G2* each gets three new clients according to *Round Robin*. At that moment there are seven clients in *G1*, and five clients in *G2*. The response time of each group increases based on their increased number of clients. *G1* has longer response time than *G2* because *G1* has two more clients than *G2*. At time slot 13, the first six clients finish running, *G1* and *G2* now each has three clients (from the second group of clients), thus the response time is reduced dramatically and almost the same for each group. At time slot 14, server *cs11* recovers. Since *G3* now has a new primary RM, *G3* becomes available to the clients again, which means there are three groups *G1*, *G2*, and *G3* available for load-balancing. But since there are no new clients coming to the system, *G3* is idle. At time slot 16, the third six clients start running before the second six clients finish running. *G1*, *G2*, *G3* each gets two new clients. *G1* and *G2* each has five clients while *G3* has two clients. The response time of *G1* and *G2* are almost the same since they both have five clients while *G3* has the lowest response time since it has only two clients. At time slot 19, the second six clients finish running in the system, and there are only six clients left in the system. *G1*, *G2*, and *G3* each has two clients now, and the response times of *G1* and *G2* decrease and are the same as *G3*.

From the experiments during the reconfiguration, we see clearly how the reconfiguration affects the performance of the system. The experiments also show clearly

the behavior of the *Round Robin* strategy. For instance, at time slot 16 in Figure 6–6, when six new clients join the system, we distribute them over the servers in a round robin fasion, i.e., each server receives two. With other mechanism, it might be possible to assign more clients to *G3* since it is currently the least loaded.

CHAPTER 7
Conclusions and Future Work

7.1 Conclusions

The current LB system provides load-balancing and performs reconfiguration automatically after failure and recovery. It is a feasible solution for the application server system.

7.2 Future work

The current LB system can still be extended to become more flexible.

1. Advanced load-balancing algorithm In the current LB system, we used *Round Robin* as the main load-balancing algorithm. However, the ideal load-balancing algorithm would be load-related. In a load-related load-balancing scheme, clients are assigned to groups based on the system load at that moment. Furthermore, the load of one client may differ from the load submitted by another client. A good load-balancing algorithm has to take this into account. The load-balancing algorithm can be easily replaced in the LB framework. Hence, the framework builds an excelled basis to study advanced load-balancing algorithms.

2. Dynamic group re-adjustment In the current LB system, an LB server can not re-read the configuration file, thus, it can not do group re-adjustment. It is currently not possible to have a new replication group in the LB system such that new sites can be added to the system and become new primaries. Thus, it is desirable to have a mechanism to allow the LB servers to re-read the configuration file and

do dynamic group re-adjustments once the configuration file changes. Instead of re-reading a configuration file, we could also add an system administration interface through which configuation changes can be submitted.

References

[1] Luis Aversa and Azer Bestavros. Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting. In *2000 IEEE International Performance, Computing and Communication Conference*, 2000.

[2] J. Balasubramanian, D. C. Schmidt, L. Dowdy, and O. Othman. Evaluating the Performance of Middleware Load Balancing Strategies. In *Eighth IEEE International Enterprise Distributed Object Computing Conference*, 2004.

[3] A. Bartoli, C. Calabrese, M. Prica, E. A. D. Muro, and A. Montresor. Adaptive Message Packing for Group Communication Systems. In *OTM Workshops 2003: 912-925*, 2003.

[4] A. Bartoli, V. Maverick, S. Patarin, J. Vučković, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Int. Symp. on Distributed Objects and Applications*, 2004.

[5] BEA Systems Inc. *BEA WebLogic Server Programming WebLogic Enterprise JavaBeans*, Release 7.0 edition, September 2002.

[6] Birman, K. P., and R. Van Renesse. *Reliable Distributed Computing with Isis Toolkit.* IEEE, 1993.

[7] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The Primary-Backup Approach. In *Distributed Systems. Second edition. ACM Press*, 1993.

[8] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems Concepts and Design.* Addison Wesley, 2001.

[9] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, 1996.

[10] Roy Friedman and Daniel Mosse. Load Balancing Schemes for High-Throughput Distributed Fault-Tolerant Servers. In *16th Symposium on Reliable Distributed Systems (SRDS'97)*, 1997.

[11] S. Frølund and R. Guerraoui. A Pragmatic Implementation of e-Transactions. In *Proc. of Symp. on Reliable Distributed Systems (SRDS)*, 2000.

[12] S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *16th ACM Symposium on Operating System Principle*, 1999.

[13] Sacha Labourey and Bill Burke. *JBoss Clustering.* The JBoss Group, 2002.

[14] Spread Concepts LLC, Center for Networking, and Distributed System (CNDS). Spread Toolkit. http://www.spread.org.

[15] V. Maverick. Object Model for Pluggable J2EE Replication Strategies. Technical report, Universitá di Bologna, Bologna, Italy, June 2003.

[16] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[17] R. Van Renesse, K.P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[18] Andreas Schaefer. JBoss: An In-Depth Look at the Interceptor Stack, 2002. http://www.onjava.com/pub/a/onjava/2002/07/24/jboss statck.html.

[19] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Symposium on Operating Systems Design and Implementation*, 2002.

[20] Scott Stark and The JBoss Group. *JBoss Administration and Development Third Edition (3.2.x Series).* The JBoss Group, August 2003.

[21] Scott M Stark and The JBoss Group. JBoss Application Server, 2002.

[22] SUN Microsystems Inc. *JAVA 2 Platform Enterprise Edition Specification, v1.3*, October 2000.

[23] SUN Microsystems Inc. *EJB 2.0 Specification*, November 2003.

[24] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specification: A Comprehensive Study. *ACM Computing Surveys*, 33(4), 2001.

[25] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, 2000.

[26] H. Wu and B. Kemme. Eager Replication Protocol for Stateful Application Servers. Technical report, McGill University, June 2004. http://www.cs.mcgill.ca/~hwu19/.

[27] H. Wu, B. Kemme, and V. Maverick. Eager Replication for Stateful J2EE Servers. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2004.