A Unified Framework for Load Distribution and Fault-Tolerance of Application Servers

Huaigu Wu[1] and Bettina Kemme[2]

[1] SAP Labs Canada, Montreal, Quebec, Canada
Huaigu.Wu@sap.com
[2] McGill University, Montreal, Quebec, Canada
kemme@cs.mcgill.ca

Abstract. This paper proposes a novel replication architecture for stateful application servers that offers an integrated solution for fault-tolerance and load-distribution. Each application server replica is able to execute client requests and at the same time serves as backup for other replicas. We propose an effective load balancing mechanism that is only load-aware if a server is close to become overloaded. Furthermore, we present transparent reconfiguration algorithms that guarantee that each replica has the same number of backups in a dynamic environment where replicas can join or leave at any time. Our evaluation shows that our approach scales and distributes load across all servers even in heterogeneous environments while keeping the overhead for fault-tolerance and load-balancing small.

1 Introduction

Application servers (AS) have become a prevalent building block in current information systems. The AS executes the application programs and maintains volatile data, such as session information, i.e., the server is *stateful*.AS often execute crucial and heavy loaded tasks, and have to handle a large number of concurrent clients, while at the same time have to provide short response times and high reliability. Both requirements can be achieved via replication. Fault-tolerant replication solutions are proposed, e.g., in [1,2,3], while scalability solutions are presented in [4]. However, little research has been performed on providing a combined solution both for scale-out and for high reliability. Similarly, existing AS products such as JBoss and WebLogic offer separate replication solutions for fault-tolerance and load-balancing each having its own setup.

In order to achieve scalability, load balancing algorithms use a cluster of AS replicas, each equipped with the same application software, and distribute request execution across the replicas. Ideally, the more replicas, the higher the maximum throughput the cluster can achieve. In this paper, we refer to a group of replicas executing requests as the *load distribution group* (LDG). In contrast, fault-tolerance algorithms use server replicas to mask the failures of individual replicas. Most commercial solutions let a primary replica execute requests, while the other replicas are backups. The primary sends data changes to backups at critical time points. If the primary fails, a *failover* procedure makes one of the backups the new primary and clients are automatically reconnected to it. For fault-tolerance purposes, it is typically enough to have one or two backups. We refer to a group of one primary and its backups as *fault-tolerance group* (FTG).

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 178–190, 2009.

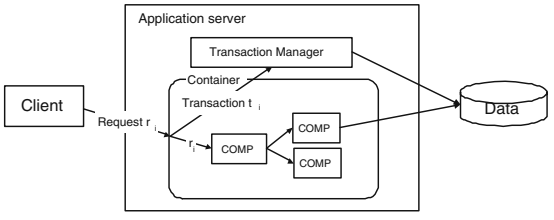


Fig. 1. Application server architecture

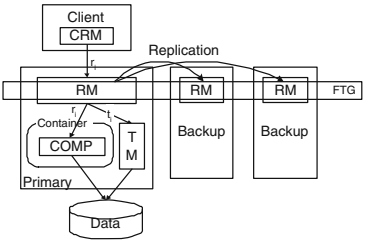


Fig. 2. Architecture of ADAPT-SIB

A simple approach to combine load balancing and fault tolerance together is to distribute the workload over a set of server replicas, each of them having dedicated backups. However, since backup tasks have typically a low overhead, capacity of the backups is wasted. Furthermore, adding a new replica to handle increased load requires to add backups, and a failed machine needs to be replaced with a new machine to handle future crashes. In order to tackle this problem, we present a unified framework where each replica executes its own client requests and at the same time is backup for other servers. All replicas of an AS cluster belong to a single LDG and can execute client requests. At the same time, each replica is primary of a (small) FTG and is backup in some other FTGs. Our unified solution distributes load across all replicas using a simple and efficient load balancing mechanism that is only load-aware if a server is threaded to become overloaded, handles failures transparently using a dynamic reconfiguration mechanism that automatically guarantees that each replica has a sufficient number of backups at any time, and can easily grow and shrink with the demands.

We use the ADAPT-SIB fault-tolerance algorithm [5,3] that provides stateful application server replication with strong consistency properties. However, our architecture is also adaptable to other fault-tolerance algorithms. We implemented our solution into the JBoss application server. Our experiments show that the approach efficiently combines fault-tolerance and scalability tasks.

2 Background

2.1 Stateful Application Server

In an AS, the business logic is modularized into different components (e.g., *Enterprise JavaBeans*) as depicted in Figure 1(a). *Volatile components* maintain session information and other related information. They are typically associated with a single client (e.g., Stateful Session Beans). Their state is lost when the AS crashes. In contrast, *persistent components* maintain data that is loaded from the databases (e.g., Entity Beans).

Client requests are calls to interface methods of components. Often, a client has to first connect to the server and after that, a session is created. The volatile components of this client are associated with this session. A client may usually only submit a request once it has received a response for its previous request. Therefore, in general, no concurrency control is needed for volatile components. When persistent components or

the backend database are accessed, requests are normally executed within a transaction that isolates the operations of concurrent transactions.

2.2 Group Communication

We use a *group communication system* (GCS) for communication where the AS replicas are the members of a GCS group. A member can multicast a message to all group members (including itself) or send point-to-point messages. The multicast we use provides *uniform-reliable delivery*. It guarantees that if any member receives a message (even if it fails immediately afterwards) all members receive the message unless they fail. Furthermore, we use multicasts that either deliver messages in FIFO order (messages of the same sender are received in sending order), or in total order (if two members receive messages m and m', they both receive them in the same order). Furthermore, GCS maintains a view of the currently connected members. Whenever the view configuration changes (through explicit joins or leaves, or due to crashes), the GCS informs all membes by delivering a view change message with the new view. Many GCS provide *virtual synchrony* [6]: If members p and q receive both first view V and then V', they receive the same set of messages while members of V.

2.3 ADAPT-SIB Replication Algorithm

The basic fault-tolerance mechanisms used in our system are based on the ADAPT-SIB algorithm [5,3]. We describe the main steps of ADAPT-SIB necessary to understand the rest of the paper. Figure 2 shows the architecture. ADAPT-SIB assumes there is one primary replica and several backup replicas. All join a single fault-tolerance group FTG (one GCS group). Each AS replica has a *replication manager* (RM) that executes the replication algorithm. At the client, there is a *client replication manager* (CRM).

The CRM has a list of all server replicas. It intercepts each client request and directs it to the current primary replica. If the primary fails, the CRM finds the new primary and resubmits the last request to the new primary if it did not receive a response before the crash. At the primary replica, each request is executed within a transaction. At commit time of a transaction, a checkpoint message containing changes performed on volatile components is multicasted with uniform reliable delivery to all replicas in the FTG. If the primary fails, all active database transactions are aborted by the database system. All remaining backup replicas in the FTG receive a view change message from the GCS and then one of them becomes the new primary. The new primary reconstructs the correct state of volatile components from the checkpoint messages. Then, the new primary receives resubmitted requests and new requests. ADAPT-SIB ensures that the state of the AS and the database are consistent and that the new primary does not execute any request twice. Previously failed or completely new replicas can join in the FTG at runtime. A joining replica receives all checkpoint messages from one existing replica in the FTG and then performs backup tasks. The cost at the backups of receiving the state changes and processing them is only 5-10% of the costs at the primary to execute the original requests. Thus, using ADAPT-SIB for fault-tolerance, each replica has the potential to act as a primary and also as backup for some others.

3 Unified Architecture

In order to exploit the resources of all replicas, we propose an architecture that allows to distribute client requests across all replicas in a cluster and at the same time guarantees fault tolerance using the ADAPT-SIB protocol. All replicas in the cluster are members of a single load distribution group LDG. Each replica is primary in exactly one fault-tolerance group FTG, referred to as its *primary FTG*, and thus, there are as many FTGs as there are replicas in the system. Furthermore, each replica is backup in m FTGs, referred to as the replica's *backup FTGs*. As a result, each FTG has m backups. We refer to this as the m/m *property*. This property allows for a simple, yet powerful automatic reconfiguration mechanism, and also helps in load distribution. In the following, we first discuss how the system starts up, then we explain how load balancing is done, and finally talk about reconfigurations.

3.1 Cluster Initialization

At initialization, one has to decide on the number m of backups in an FTG. Furthermore, one has to indicate the initial number n of replicas in the cluster. The cluster size may shrink or increase later dynamically, but m remains fixed.

Figure 3 shows the final setup after initialization. Each replica uses its address as a unique identifer. At each replica LBM refers to a *Load Balancing Manager* (LBM), PRM to the primary replication manager of the primary FTG, and $BRMS$ refers to the array of backup replication managers for the backup FTGs. When a replica starts up, its LBM first joins the LDG. Once all n replicas have joined, each LBM multicasts its replica identifier using total-order, uniform-reliable delivery. Each LBM receives the messages in the same order and stores the identifiers in an ordered list, called the replica list RL according to the delivery order. Each replica in the system is assigned an *order* number i, $1 \leq i \leq n$, which is the position of the replica in the replica list RL. We refer to the replica with order number i as r_i. Note that while the identifier of a replica does not change during its lifetime, the order number might change, as we will see later. Once r_i has determined its order number i, it joins fault-tolerant group FTG_i as primary. If $i > m$, it furthermore joins FTG_{i-m} to FTG_{i-1} as backup. A replica with order $i \leq m$ joins FTG_{n-m+i} to FTG_n and FTG_1 to FTG_{i-1} as backup. For

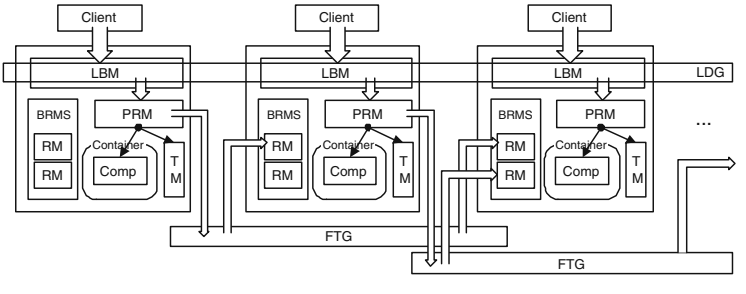


Fig. 3. Unified Architecture

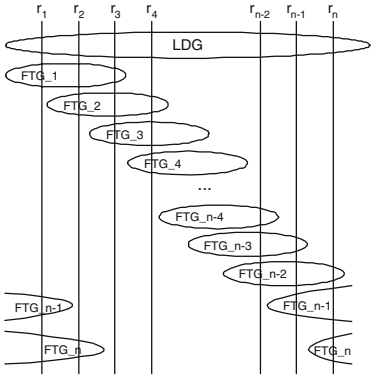


Fig. 4. Cluster Initialization

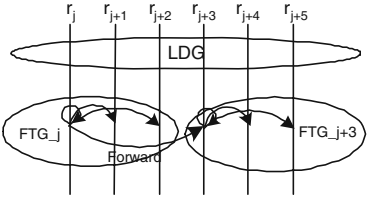


Fig. 5. Forwarding a request

instance if $m = 2$, then r_3 joins FTG_1 and FTG_2, r_2 joins FTG_n and FTG_1, and r_1 joins FTG_{n-1} and FTG_n. Figure 4 depicts this circular setup of FTGs.

3.2 Load Balancing

Simple Load Distribution. When a client connects to the system, a session on one primary replica is created, and all requests within this session are handled by this replica. This guarantees that each request sees the current session state. Thus, load balancing is performed at a connection time. It goes through two phases. The client has a predefined replica list CL with potential server replicas (received, e.g., when looking up the service on a registry). It submits connection requests to these replicas until it receives a response. When an available replica r_i receives such a request it becomes the load-balancer for this request. The LBM of r_i randomly decides on a replica r_j from its replica list RL to serve the client, and returns the client replication manager CRM object to the client. The message piggybacks a failover list containing the identifiers of all members of FTG_j (derived from replica list RL) and indicates that r_j is the primary. The failover list FL is used by the client CRM for fault tolerance, and is described in more detail in Section 3.3. In the second phase, the newly installed CRM sends a session request to r_j. Then r_j accepts client requests within the session.

Our load-balancing algorithm is very efficient, as the message overhead for the session setup is the same as in standard AS architectures involving one message round for the connection request (with r_i), and one for session creation (with r_j). Scheduling decisions are made locally at the server site without the need to exchange and maintain load information. The replica list CL does not need to be accurate or complete as the sessions themselves are equally distributed among all active replicas.

Load Forwarding. Random replica assignment, however, does not work well in heterogeneous environments or if request execution times are not uniform. For example, if a server gets a few heavy requests it will be temporarily overloaded and any further request will deteriorate the performance. We address this issue with a simple but effective

first-local-then-forward (FLTF) mechanism which leads to request forwarding if the load of a replica is above a given threshold. Load could be measured as memory usage, CPU usage, response time, or the number of connected clients. We refer to a replica with a load below the threshold as a *valid* replica.

When the LBM of replica r_j receives a client session request and its load is below the threshold, it serves the client directly as described above. Otherwise, r_j multicasts a load query message lqm to all replicas in FTG_j in order to find a valid replica. (see left ellipse in Figure 5). Upon receiving lqm a backup returns a positive answer if its load is below the threshold. r_j chooses the first r_k to answer as the one to serve the client. It returns to the CRM the list of replicas belonging to FTG_k. The CRM refreshes its local failover list FL, and sends a session request to r_k. In case of isolated overloads, contacting m other nodes will likely find a node that can accept new clients. However, if there is no positive answer among the backups, r_j sends a *forward* message to the replica with the smallest order number larger than any order number in FTG_j, i.e., $r_{(j+m+1)\%n}$. $r_{(j+m+1)\%n}$ now repeats the process, sending a new load query message lqm in its own primary $FTG_{(j+m+1)\%n}$. Figure 5 shows how a forward is sent to the primary of FTG_{j+3} if $m = 2$. If a valid replica is found, $r_{(j+m+1)\%n}$ returns the information to r_i so that it can forward the relevant replica list to the client. Otherwise, an additional forward could take place. If after a maximum number of T iterations no valid replica is found, a negative message is sent to the originator r_j which either accepts or refuses the client. If T is set 0, then no forward message is sent at all. Setting T low makes sense because if all nodes in r_j's neighborhood are saturated, it is likely that the entire system is close to saturation and further forwarding will not help.

Discussion. A main benefit of our load distribution algorithm is that it is purely distributed without any central controller and can be easily implemented. Load messages are only sent if a node is overloaded, load is checked in real time, and replicas can individually decide to take on further load or not. It does not affect the fault-tolerance algorithm but takes advantage of the FTG infrastructure.

One question is how often one has to forward to find a lightly loaded replica. The probability of finding a valid replica within the m backup replicas of the local FTG is equal to the probability of finding a valid replica within any m replicas in the cluster. If there are k valid replicas randomly distributed in the cluster, the probability p of finding one of the k valid replicas within the m backup replicas is $p = 1 - \binom{n-m-1}{k} / \binom{n-1}{k}$. Since each forward searches $m+1$ replicas (a new FTG), the probability p of finding one of k valid replicas within the m backups of the initiator and T further forwards is $p = 1 - \binom{n-m-1-T*(m+1)}{k} / \binom{n-1}{k}$. Assume the cluster has 100 replicas and $m = 2$. With 50 valid replicas and $T = 1$ we find a valid replica with more than 97% probability. With only 20 valid replicas and $T = 3$, the probability is 92.8%. Thus, this simple mechanism provides a high success rate even for highly loaded clusters with low message overhead.

3.3 Reconfiguration

In this section, we show how the system automatically maintains the m/m property (each FTG has m backups, each replica is backup of m $FTGs$) which guarantees that each replica has at any time enough backups.

Server crash. When a replica r_i fails it leaves FTG_i, and a new primary has to be found for r_i's clients. Furthermore r_i is removed as a backup from m FTGs. Thus, these FTGs need new backups. For simplicity of notation the following discussion assumes that a replica r_i fails where $i > m$ and $i + m < n$.

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

- r_{i+1}, which is a backup in FTG_i, becomes the new primary for r_i's clients. It first performs the failover for the sessions of clients in FTG_i as described in ADAPT-SIB. Then, it makes these clients part of FTG_{i+1}. Finally, it leaves FTG_i.
- The other backups in FTG_i (r_{i+2} to r_{i+m}) must also migrate the session information of the old clients of FTG_i to FTG_{i+1}. After the migration, they also leave FTG_i. As a result, FTG_i does not have any member anymore.
- r_{i+m+1} was not member of FTG_i but now needs to receive the session information of the clients that were migrated from FTG_i to FTG_{i+1}. The primary of FTG_{i+1} sends this information to r_{i+m+1}.
- Since replicas r_{i+1} to r_{i+m} have left FTG_i, they are now backups for only m-1 FTGs. Furthermore, FTG_{i-m} to FTG_{i-1} only have m-1 backups since r_i was removed from these groups. To resolve this, r_{i+1} joins FTG_{i-m} as backup, r_{i+2} joins FTG_{i-m+1}, etc, as the joining process of new replicas described in ADAPT-SIB.
- Finally, each load balancing manager LBM removes r_i from its replica list RL, and decreases the order numbers of replicas r_{i+1} to r_n by one.

Server recovery. When a new or failed replica joins in a cluster of size n, it first joins the single load distribution group LDG, and all replicas are notified about this event. Each replica adds the new replica with order number $n + 1$ to its replica list RL and considers it in its load-balancing task. r_{n+1} receives the replica list RL from a peer replica. According to our setting, r_{n+1} must have a primary FTG and m backup FTGs.

- r_{n+1} creates a new FTG_{n+1} and joins it.
- r_{n+1} joins FTG_{n-m+1} to FTG_n as backups. These FTGs have now $m + 1$ backups.
- Now, r_1 to r_m leave FTG_{n-m+1} to FTG_n respectively. The FTGs are now back to having m backups.
- Finally, r_1 to r_m join the new FTG_{n+1}. They have again m backup FTGs and FTG_{n+1} has m backups. The recovery is fast, since this group is new.

The reconfiguration is complete and r_{n+1} starts accepting client requests.

4 Experiments and Evaluation

We integrated our approach into the JBoss application server (v. 3.2.3) [7]. For this paper, we use a micro benchmark where each client request performs operations on stateful session beans associated with the client but the database is not accessed. This allows us to isolate the replication effects on the AS. Clients connect to the system

and then run for 10 seconds continuously submitting requests. All requests trigger transactions with similar load. Further experiments can be found in [8]. Unless otherwise stated, experiments were performed on a cluster of 64-bit Xeon machines (3.0 GHz and 2G RAM) running RedHat Linux. As GCS we use Spread [9]. In all our settings, each FTG consists of one primary and two backups.

4.1 Experiment 1: Basic Performance

We have a first look at the performance of our unified architecture when no replicas leave or join the system. In the figures, `JBoss` refers to a standard single-node non-replicated JBoss application server without fault-tolerance. `ADAPT-SIB` refers to a system running the ADAPT-SIB algorithm, i.e., there is one fault-tolerance group FTG but no load distribution group LDG. `ADAPT-LB` refers to the approach proposed in this paper with one LDG using our load-balancing approach and several FTGs running ADAPT-SIB. `JC/RoundRobin` refers to a replicated cluster using JBoss' own round-robin based load-balancer. This configuration does not provide any fault-tolerance.

Figure 6 shows response times with increasing number of clients injected in the system per second for three machines. In the non-replicated JBoss and ADAPT-SIB clients compete soon for resources and response times deteriorate quickly. ADAPT-SIB has higher response times than a non-replicated JBoss, since the primary has to perform the replication. In contrast, ADAPT-LB and JC/RoundRobin have low response times up to 15 clients due to load distribution. Each node is less loaded and can provide faster service. While ADAPT-LB has higher response times than JC/RoundRobin the difference is smaller than between ADAPT-SIB and the non-replicated JBoss, because ADAPT-LB is able to distribute the fault-tolerance overhead across all replicas.

Scalability is further analyzed in Figure 7 which shows that the throughput for ADAPT-LB and JC/RoundRobin increases linearly with the number of replicas. Due to fault-tolerance activity on each node, ADAPT-LB achieves less throughput than JC/RoundRobin. But even JC/RoundRobin does not provide ideal throughput since load-balancing has its own overhead.

In summary, ADAPT-LB truly provides both fault-tolerance and scalability.

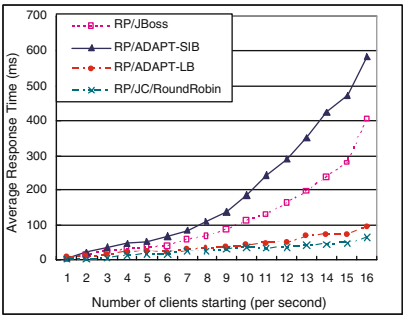


Fig. 6. Response Times with 3 Replicas

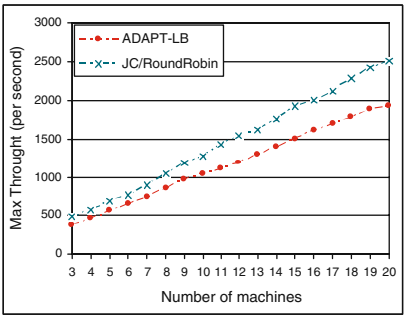


Fig. 7. Scale-up

4.2 Experiment 2: Heterogeneity

Heterogeneity is a challenge for load-balancing techniques. We first analyze the impact of heterogeneous hardware by replacing half of the machines with PIII machines (850 MHz and 256M RAM). Figure 8 shows the maximum achievable throughput with increasing number of machines for ADAPT-LB with forwarding (`ADAPT-LB/FLTF`), ADAPT-LB using only random load-balancing without forwarding (`ADAPT-LB/Random`), and JBoss' round-robin load-balancer (`JC/RoundRobin`). In general, the throughput is lower than in the homogeneous environment (Fig. 7), since half of the machines are now weaker. ADAPT-LB/Random is the worst because random assignment ignores heterogeneity and fault-tolerance adds overhead. ADAPT-LB/FLTF and JC/RoundRobin have similar performance despite the fact that ADAPT-LB has the fault-tolerance overhead. A detailed analysis has shown that, compared to JC/RoundRobin, ADAPT-LB/FLFT has lower throughput on the weak and higher throughput on the strong nodes. This is because with ADAPT-LB/FLFT weak nodes forward requests that are then executed by the strong nodes. Thus, ADAPT-LB/FLTF compensates the overhead of fault-tolerance by a smarter load-balancing strategy which assigns more tasks to the stronger nodes.

Another type of heterogeneity are dynamic workload changes. In the next test, we use an additional very heavy client transaction with an average response time of 2000 ms to simulate the heterogeneous workload. We only compare ADAPT-LB/FLTF and JC/RoundRobin. At the beginning of this test, a cluster consisting of 6 identical machines runs the micro benchmark for 30 seconds. Then we artificially inject the heavy transaction into the system. We refer to the machine executing the heavy transaction as HC. The other are denoted as LC. Figure 9 has as x-axis time slots of 100 ms. The heavy transaction starts at timeslot 5. Before injecting the heavy transaction, HC and LC have the same response times which are higher for ADAPT-LB because of the fault-tolerance overhead. Using JC/Round-Robin, the HC response times increase to around 400 ms after the injection because the HC machine becomes saturated. The response times on the LC group remain the same because they are not affected. Using ADAPT-LB, response times on the HC machine increase for the first 5 time slots after the heavy transaction is injected. This represents transactions of clients that were already

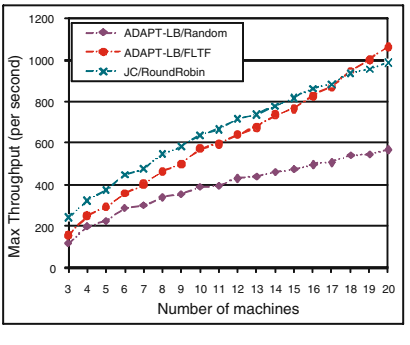


Fig. 8. Heterogeneous hardware

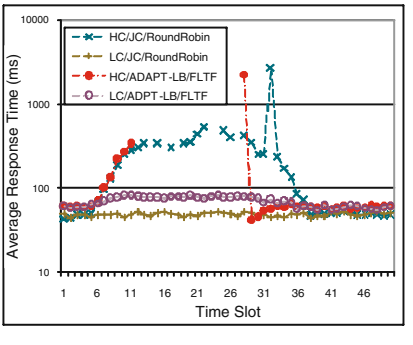


Fig. 9. Heterogeneous workload

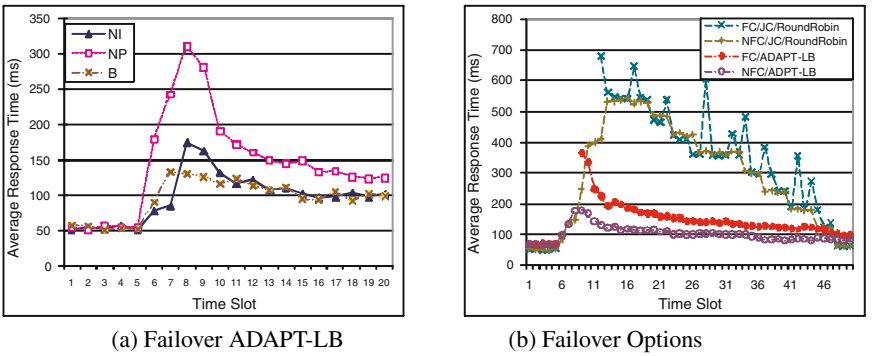
assigned to HC when the long transaction arrived. Then there is a long gap, since HC does not accept any further clients anymore according to the forwarding strategy. At timeslot 27 the long transaction finishes (with a long response time). After that HC accepts clients providing standard response times for them. While the heavy transaction is running on HC we observe longer response times at the LC machines because HC redirects clients to them, and thus they are more loaded. In total, response times are less affected with ADAPT-LB than with JC/RoundRobin which has unacceptable high response times for some of the clients.

Clearly, our approach can quickly respond to the temporary overload of individual machines. The extra overhead only has to be paid when overload occurs.

4.3 Experiment 3: Failover and Recovery

We now show the behavior of the system during and after reconfiguration. A cluster of 6 replicas ($r_1, ..., r_6$) runs the micro benchmark for about 30 seconds when replica r_3 crashes. We distinguish three types of replicas. NP (new primary) indicates the replica r_4 that takes over the clients of the failed replica r_3. B indicates replicas that have to reconfigure their backups (r_5 and r_6). NI indicates all other replicas on which the failure has no impact (r_1 and r_2). Figure 10(a) has as x-axis time slots of 100 ms, and as y-axis the average response time within a time slot. The crash occurs at time slot 5.

Before the crash, the average response time is similar in each group. After the crash, the response time on the new primary NP drastically increases because it requires considerable resources to perform the failover. This process takes about 300-400 ms. After that, the average response time is still higher than on the other groups because NP has now double the clients. The response time in the replicas that have to reconfigure their backups (B) also increases (it shortly doubles) because the state transfer to the new backups takes some of the resources. The recovery process to become a backup takes less than 100 ms. However, the response time on B remains higher and actually also increases on NI for which no reconfiguration is necessary. The reason is that there is now one less replica in the system to execute requests. Furthermore, since NP is still higher loaded, the replicas in B and NI accept more of the newly injected clients.



(a) Failover ADAPT-LB (b) Failover Options

Fig. 10. Reconfiguration

Eventually, once NP has stabilized, the system becomes balanced again. The average response time on all remaining replicas converges eventually to the same value. This value is now higher because there is one less replica in the system to serve requests.

We further conducted a test where we added a 7th replica to a running system. The join shortly doubled the response times of those clients that were connected to replicas who had to change their $FTGs$. But the system reconfigured in less than 400 ms. After that response times are generally lower as load was now distributed over more replicas.

As a final experiment, we compare ADAPT-LB with an alternative solution. Using JBoss' round-robin architecture, when a replica crashes, each client originally connected to the failed replica connects to any of the correct replicas and resubmits all requests from the beginning of the session. We call this solution the *re-execution solution*. Note that this only works correctly if the requests do not trigger changes on permanent components because these changes are already in the database and should not be applied again. We again use 6 machines and crash one replica at time slot 5. This time, we group response times by client type. FC clients were originally connected to the crashed replica, and NFC are all other clients. Figure 10(b) shows the average response time over time. In ADAPT-LB, one replica takes all FC clients (and has additionally NFC clients). As long as this new primary performs failover, the FC are blocked. Therefore, there is a gap for FC clients where no response times are measured, and once execution resumes there is a peak in average response times. NFC clients are also affected, but much less (as discussed before). Response times for both FC and NFC quickly go back to normal levels. As long as the new primary is more loaded, the forwarding strategy distributes new clients to other replicas.

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

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

5 Related Work

Load balancing and fault-tolerance have traditionally be handled as orthogonal issues, and research on one topic usually does not attempt to solve the other. For fault tolerance of application servers, most industrial (e.g., JBoss, Weblogic or WebSphere) and many research solutions (e.g., [2,3] use the primary-backup approach.

Typical load balancing solutions of application or web servers use a central load balancer. As we have seen, content-blind policies [10], such as Random or Round Robin do not work well in heterogeneous environments. Content-aware policies (e.g., [4,11,12]

require some knowledge about the environment, such as load or access patterns which can have considerable overhead. In contrast, our approach is purely distributed, does not maintain state information, has little overhead, and is easy to implement.

Only a few approaches consider both load distribution and fault-tolerance. Singh et al. [13] propose a system that merges the Eternal fault tolerance architecture [1] and the TAO load balancer [4]. A similar architecture is used in [14]. All servers in a cluster are partitioned to several disjoint FTG groups. Only the primary server in each replica group is used for load balancing. Moreover, these solutions do not address reconfiguration problems. [15] organize replicas in a chain and execute updates on the head (who then propagates the changes down the chain) while reads are executed on the tail. However, client requests are distributed over head and tail which his problematic for stateful application servers. Also, not all replicas are evenly utilized.

Scalability and fault-tolerance is also an issue in file-systems. For instance, Coda [16] also distributes load over servers and uses replication for availability. But files are always persistent and there is no backend database. Also, Coda does not follow the primary-backup replication but clients take care to update all copies.

6 Conclusion

This paper describes a new replication architecture for stateful application servers that offers both fault-tolerance and load balancing in a single integrated solution. Replication is completely transparent to the clients, all resources in the system are used, and the system requires little intervention by administrators. The solution is simple to implement yet powerful. The architecture is completely distributed. Our implementation shows that our approach increases the scalability even in heterogenous environments, and provides dynamic reconfiguration in a dynamic environment with little overhead.

References

1. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Strongly consistent replication and recovery of fault-tolerant CORBA applications. Journal of Computer System Science and Engineering 32(8) (2002)
2. Barga, R., Lomet, D., Weikum, G.: Recovery guarantees for general multi-tier applications. In: Int. Conf. on Data Engineering, ICDE (2002)
3. Wu, H., Kemme, B.: Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In: IEEE Symp. on Reliable Distrib. Systems, SRDS (2005)
4. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Computer Communications 21(4) (1998)
5. Wu, H., Kemme, B., Maverick, V.: Eager replication for stateful J2EE servers. In: Int. Symp. on Distributed Objects and Applications, DOA (2004)
6. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Computing Surveys 33(4) (2001)
7. Fleury, M., Reverbel, F.: The JBoss extensible server. In: Middleware (2003)
8. Wu, H., Kemme, B.: A unified framework for load distribution and fault-tolerance of application servers. Technical Report SOCS-TR-2009.1, McGill University (2009)
9. Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins Univ. (2004)

10. Andreolini, M., Colajanni, M., Morselli, R.: Performance study of dispatching algorithms in multi-tier web architectures. SIGMETRICS Performance Evaluation Review (2002)
11. Pai, V.S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-aware request distribution in cluster-based network servers. In: Int. Conf. on Architectural Support for Programming Languages and Operating Systems (1998)
12. Liu, X., Zhu, X., Padala, P., Wang, Z., Singhal, S.: Optimal multivariate control for differentiated services on a shared hosting platform. In: IEEE Conf. on Decision and Contr. (2005)
13. Singh, A.V., Moser, L.E., Melliar-Smith, P.M.: Integrating fault tolerance and load balancing in distributed systems based on CORBA. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 154–166. Springer, Heidelberg (2005)
14. Othman, O., Schmidt, D.C.: Optimizing distributed system performance via adaptive middleware load balancing. In: ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (2001)
15. Renesse, R.V., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: Symp. on Operating System Design and Implementation, OSDI (2004)
16. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Trans. Computers 39(4), 447–459 (1990)