Integrating Fault Tolerance and Load Balancing in Distributed Systems Based on CORBA[*]

A.V. Singh, L.E. Moser, and P.M. Melliar-Smith

Department of Computer Science,
Department of Electrical and Computer Engineering,
University of California, Santa Barbara, CA 93106
avsingh@cs.ucsb.edu
{moser, pmms}@ece.ucsb.edu

Abstract. Fault tolerance and load balancing middleware can increase the quality of service seen by the users of distributed systems. Fault tolerance makes the applications more robust, available and reliable, while load balancing provides better scalability, response time and throughput. This paper describes a software infrastructure that integrates fault tolerance and load balancing within a distributed system based on CORBA. The software infrastructure employs Eternal's FTORB, which replicates CORBA applications and thus makes them fault tolerant, and TAO's Load Balancer, which balances the load of the clients' connections across multiple instances of a CORBA server.

1 Introduction

As distributed applications are deployed more widely, the need for improved scalability, response time and throughput becomes more important. An effective way to address this need is to employ a load balancer, based on distributed object middleware, such as the Common Object Request Broker Architecture (CORBA). Also important is the availability, reliability and robustness of the services that the applications provide and, thus, fault tolerance is essential. Fault tolerance employs replication to mask faults and provide continuous service to the users.

Fault tolerance and load balancing can be thought of as orthogonal aspects of quality of service. Both fault tolerance and load balancing require the availability of multiple computers, which distributed systems provide. Many industries with mission-critical applications, such as telecommunications, financial, aerospace and defense, need both fault tolerance and load balancing within a single integrated infrastructure.

The Object Management Group has developed specifications for fault tolerance [7] and load balancing [8] based on CORBA. In this paper, we describe a software infrastructure that integrates fault tolerance and load balancing for CORBA-based distributed systems. We discuss challenges that we had to address in the integration, and present performance results that we obtained for the integrated infrastructure.

[*] This research has been supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

2 The Fault Tolerance and Load Balancing Infrastructure

Our fault tolerance and load balancing infrastructure is based on Eternal Systems' FTORB and TAO's Load Balancer, both of which support the industry-standard CORBA distributed object computing standard. First, we describe each of those middleware components and, then, we describe our fault tolerance and load balancing infrastructure, which integrates those components.

2.1 Eternal's FTORB

Eternal's FTORB [6] provides fault tolerance by replicating servers, clients and infrastructure components (*e.g.*, the Load Balancer) that are implemented as CORBA objects, using active or semi-active replication. FTORB replicates the state of an object in volatile memory, rather than on stable storage, to eliminate the time required to write (read) that state to (from) disk. Research on integrating replication with transactions and databases, where state is persisted to disk, can be found in [13].

FTORB is not itself an ORB but is middleware that works with CORBA ORBs that support the Internet Inter-ORB Protocol (IIOP) running over TCP/IP, with no modification to the ORB. IIOP allows clients and servers that run over different ORBs to communicate with each other. Fig. 1 shows the components of Eternal's FTORB middleware. The functionality of each component is described below:

- **Interceptor:** The Interceptor Library, coupled with each client or server in a fault tolerance domain, intercepts messages and diverts them to the Eternal Replication Engine and the Totem Multicast Protocol, instead of sending them over TCP/IP.
- **Replication Engine:** The Replication Engine maintains groups of client and server replicas, and interacts with the Interceptor Library.
- **Totem Multicast Protocol:** The Totem Multicast Protocol [5] interacts with the Replication Engine and multicasts the clients' requests and servers' replies to the client and server groups, using a logical token-passing ring. Totem can be replaced with any other reliable totally-ordered multicast protocol, such as the Rose rotating sequencer protocol which is also deployed with FTORB.

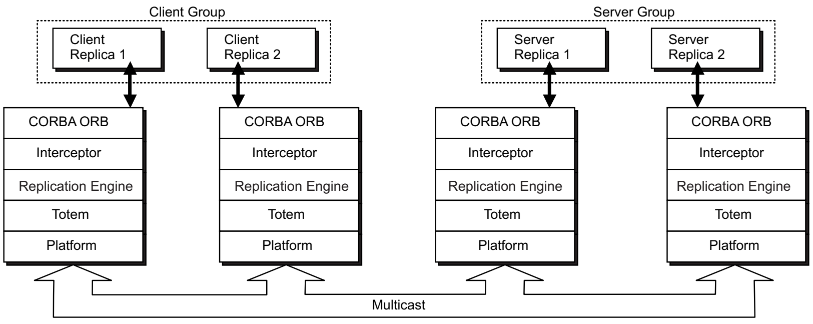

Fig. 1. The components of Eternal's FTORB

The FTORB components interwork as follows. A client group, inside the fault tolerance domain, issues a request to a server group inside the same fault tolerance domain. The Interceptor Library intercepts the request and passes it to the Replication Engine. The Replication Engine receives the request and, instead of forwarding it to the designated server via TCP/IP, forwards the request to Totem, which multicasts it to the server group. The replicas in the server group process the client's request and forward the reply to Totem, which multicasts the reply to the client group. FTORB detects and suppresses duplicate requests and replies from the replicas of the clients and the servers.

2.2 TAO's Load Balancer

TAO's Load Balancer uses a middleware load balancing approach that works with the TAO CORBA ORB [11]. TAO uses the terms Replica Locator and Replica Proxy for two of its modules. If the server that is being load balanced is stateless, those terms are fine. If the server is stateful, the instances of the server that serve different clients are typically not replicas and, thus, we refer to them as peers, instead of replicas. Correspondingly, we use the terms Peer Group, Peer Locator and Peer Proxy for the instances of the server that provide load balancing (see Fig. 2). Use of this terminology avoids confusion between the instances (replicas) of an object that provide fault tolerance and the instances (peers) of an object that provide load balancing.

- **Peer Locator:** The Peer Locator identifies which peer server will receive a client's request and binds the client to the identified peer server. The Peer Locator forwards each request it receives to the peer server selected by the Load Analyzer.
- **Load Analyzer:** The Load Analyzer determines which peer server will receive a client's request and decides when to switch loads between peer servers.
- **Load Monitor:** For a given peer server, a Load Monitor monitors the load on that peer, reports peer loads to the Load Analyzer and responds to load advisory messages from the Load Analyzer.
- **Peer Proxy:** Each object managed by the Load Balancer communicates with the Load Balancer via a Peer Proxy. The Load Balancer uses the Peer Proxies to distinguish between different peer servers.
- **Load Balancer:** A collective term for all of the above components.

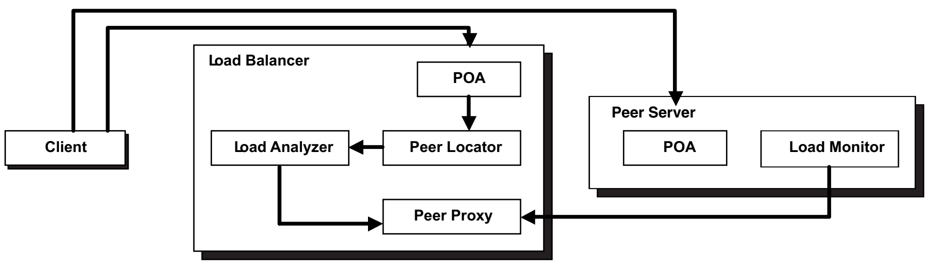

Fig. 2. The components of TAO's Load Balancer

The TAO Load Balancer components interwork as follows. A client obtains an object reference to what appears to be a peer server and issues a request. In actuality, the client invokes the request on the Load Balancer. The Portable Object Adapter (POA) of TAO dispatches the request to the Peer Locator. The Peer Locator queries the Load Analyzer for an appropriate peer server and sends back a LOCATION_FORWARD to the client to redirect the client to the selected peer server. From then on, the client sends its requests directly to that peer server. The Load Analyzer continuously communicates with the Load Monitor for the peer server. If the Load Analyzer finds the peer server to be overloaded, it issues a load advisory to the Load Monitor for that peer server. On receiving the load advisory from the Load Analyzer, the Load Monitor issues a message to the peer server, telling it to accept or redirect the next request.

2.3 Integration of Eternal's FTORB and TAO's Load Balancer

Fault tolerance and load balancing are two orthogonal aspects of quality of service. We consider the composition of these two non-functional properties and the product of peer groups (used for load balancing) and replica groups (used for fault tolerance) in the integrated infrastructure, as shown in Fig. 3.

A peer group, maintained by the Load Balancer, consists of one or more peer servers. Different peer servers handle different clients' requests. The Load Balancer distributes the requests of the different clients across the peer servers in a peer group, in order to balance the load across the peer servers. Each peer server in a peer group has one or more replicas for fault tolerance.

A replica group, created by the FTORB fault tolerance infrastructure, consists of one or more replicas that provide protection against faults. Each peer server is a member of a distinct replica group. With active replication, the replicas of a particular peer server handle the same clients' requests and have the same load.

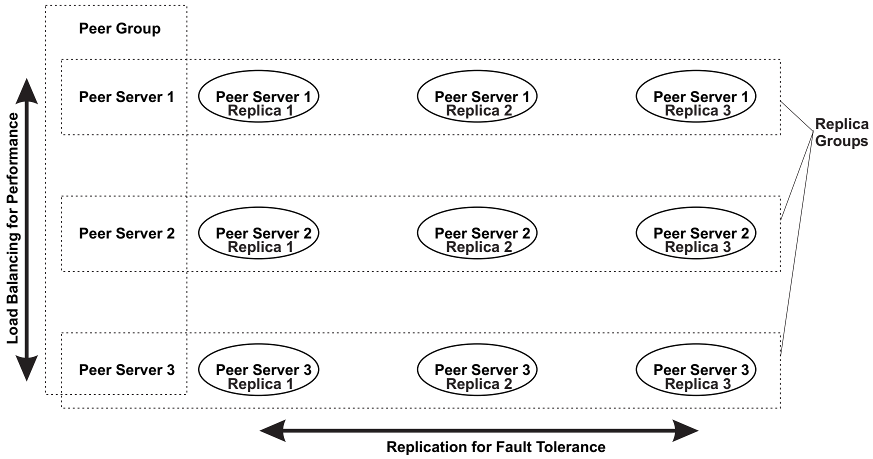

Fig. 3. The product of the peer groups, used for load balancing, and the replica groups, used for fault tolerance, in the integrated infrastructure

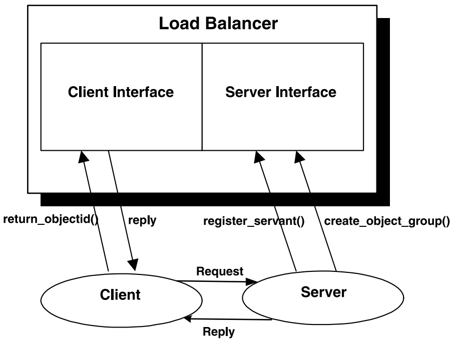

Fig. 4. Interfaces of our Load Balancer

In the integrated infrastructure, not only the peer servers but also the Load Balancer and the clients can be replicated using Eternal's FTORB to provide fault tolerance, so as to avoid a single point of failure. If, during the execution, one of the replicas fails, another replica continues the operations. FTORB then obtains the application state from an existing replica (*i.e.*, takes a checkpoint), and supplies that state to a new or recovering replica, in order to maintain the required level of redundancy.

When a stateful server serves multiple clients, and maintains different state or shared state between them, static load balancing (*i.e.*, directing a client's requests to a particular peer server for an entire session) is appropriate. Dynamic load balancing (*i.e.*, transferring a client's connection from one peer server to another in the midst of a session) is more challenging, because the state of the particular client held by the peer server (rather than the entire state of the peer server) must be transferred from one peer server to another. Our infrastructure provides fault tolerance for stateful servers and uses static load balancing.

In our infrastructure, the Load Balancer has two interfaces, the Server Interface and the Client Interface, as shown in Fig. 4 and described below.

– **Server Interface:** The server interface exposes methods that are invoked by the servers. These methods include *create_object_group()* and *register_servant()*, which enable the peer servers to create a peer group and register with the Load Balancer. After the peer servers have created a peer group and registered with the Load Balancer, the Load Balancer has a clear view of the peer servers that are available. The Load Balancer is then ready to balance the loads across the peer servers within the peer group, on receiving requests from the clients.
– **Client Interface:** The client interface exposes methods that are invoked by the clients. These methods include the *return_objectid()* method, which enables a client to obtain an object group reference for a peer server's replica group. On receiving a request from a client for an object group reference corresponding to a particular object id, the Load Balancer looks in its table to find an appropriate reference, using its particular load balancing policy to select a peer server, and then returns the object group reference of the peer server's replica group to the client. On obtaining the reference, the client issues requests to the peer server, which are multicast to the peer server's replica group.

Fig. 5 shows the use of our infrastructure in an example configuration consisting of the Load Balancer with two replicas each, two peer servers with two replicas each, and three clients with one replica each. The dashed boxes represent the replica groups. Each request (reply), instead of being sent to a single server (client), is multicast to a server (client) replica group. For each peer group, the load balancing takes place across the two peer servers that constitute that peer group.

First, the replicas of the Load Balancer are brought up on different processors. The first replica invokes FTORB to create a replica group with itself as a member, and then the second replica adds itself to the replica group.

Next, the replicas of the peer servers are brought up on different processors. Each peer server invokes FTORB to create a replica group with itself as a member and registers with the Load Balancer, using a multicast request (arrows 1 and 2). The Load Balancer creates a peer group, and adds each peer server to the peer group. The subsequent replicas of each peer server add themselves to the replica group of that peer server.

Then, the clients are brought up on different processors. Each client invokes FTORB to create a client replica group, consisting of one member each. In the example, client 2 invokes the Load Balancer's *return_objectid()* method to request a reference for a peer server, using a multicast request (arrow 3). Depending on its load balancing policy, the Load Balancer replies with the object group reference of the replica group of one of the peer servers, using a multicast reply (arrow 4), in this case, peer server 1.

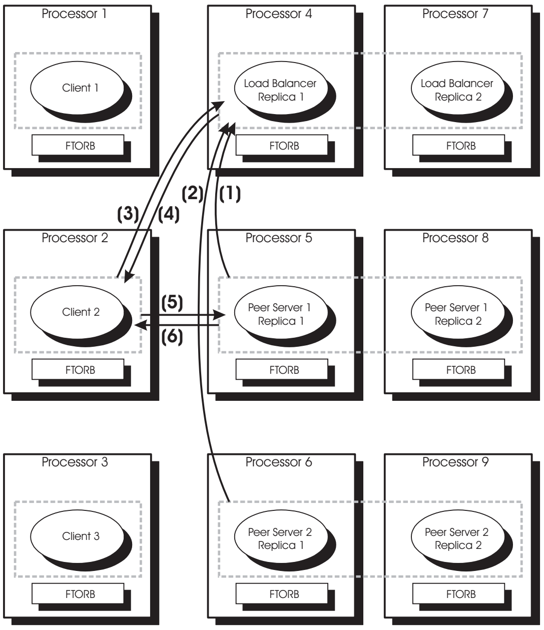

Fig. 5. The Load Balancer and the two peer servers are actively replicated, with two replicas each, using FTORB to provide fault tolerance. For each peer group, the load balancing takes place across the two peer servers that constitute that peer group

After receiving the peer server reference, client 2 invokes methods directly on peer server 1, using the object group reference of that peer server's replica group and a multicast request (arrow 5). The replicas of peer server 1 generate the response and multicast the reply to client 2, using a multicast reply (arrow 6).

3 Challenges and Their Solutions

Now we discuss the challenges that we faced in designing and implementing the integrated infrastructure and our solutions to those challenges. The challenges can be divided into two categories:

– Challenges faced in designing the Load Balancer
– Challenges faced in designing the client and server applications.

3.1 Challenges in Designing the Load Balancer

First we discuss challenges that we faced in designing the Load Balancer.

Challenge 1: TAO's Load Balancer provides load balancing by intercepting the clients' requests and returning a LOCATION_FORWARD, which contains the reference to the selected peer server. When the client receives the LOCATION_FORWARD, it sends its subsequent requests to that peer server.

To intercept requests, the Load Balancer uses the Servant Manager in a Portable Object Adapter (POA) that has USE_SERVANT_MANAGER and NON_RETAIN policies. Servant Managers are used to activate servants dynamically. Servant Managers have two interfaces, the ServantActivator and ServantLocator interfaces. TAO's Load Balancer uses the ServantLocator interface to intercept a client's request and instantiate a servant for it on the fly.

This approach works fine for load balancing, but does not work when integrated with fault tolerance. The reason is that Eternal's FTORB recognizes replicas only if they have the same object ids. Because the POA's policy is set to NON_RETAIN, the object ids of the activated server objects are not persistent and, hence, are not recognized as replicas by FTORB.

Rejected Strategies: It was not possible to solve this problem by changing the POA's policies, because the USE_SERVANT_MANAGER and NON_RETAIN policies were required to use the ServantLocator interface.

First, we considered dividing the functionality of TAO's Load Balancer into two parts. One part would interact with the servers and the other part with the clients. This solution was not feasible for the reason that it required that one POA exposes methods for the servers and has the PortableServer::PERSISTENT and PortableServer::USED_ID policies, and the other POA exposes methods for the clients and has the USE_SERVANT_MANAGER and NON_RETAIN policies. However, the POA with the ServantLocator interface is unaware of server objects that are active inside the other POA. Moreover, this solution would misuse the ServantLocator, as servant objects are already active and, thus, do not require the services of the ServantLocator.

Another possible solution was to write our own interceptors using CORBA's portable interceptors, which replace the ServantLocator interface in the Servant Manager. Portable interceptors expose pre-defined interception points, at both the client and the server, which make the requests and replies accessible. In TAO's Load Balancer, we intercepted the client's request, so the interception points are *receive_request_service_contexts()* and *receive_requests()* on the server-side. The problem was that, when the request was intercepted at the *receive_request_service_contexts()* interception point, we did not have access to the object id, as operation parameters were not available. Thus, it was impossible to determine the server group to which the request was addressed. Before the request reached the *receive_requests()* interception point, where the parameters were accessible, it was too late. The reason is that, after *receive_request_service_contexts()*, the servant manager was invoked, but as the servant manager was not activated, a system exception was raised. Thus, using CORBA's portable interceptors did not work.

Solution: The solution to this problem is to have the clients and servers communicate directly with the Load Balancer. The clients and servers are responsible for resolving the reference to the Load Balancer, and issue explicit requests to it. By adopting this solution, we avoided using both the ServantLocator and CORBA's portable interceptors.

In this approach, the servers are activated inside a POA, having the PortableServer:: PERSISTENT and PortableServer:: USED_ID policies. Because these policies are set, the object group references, which are written to IOR files, are persistent. As their object ids are persistent, FTORB can recognize the server replicas. The client does not send its first request to the Load Balancer, but resolves the reference to the Load Balancer and queries the Load Balancer for the peer server to which to connect. In the previous approach, the Load Balancer returns the object group reference for the peer server's replica group to the client, which also happens with this approach. The difference is that now the client explicitly requests the object group reference for the peer server's replica group from the Load Balancer.

Challenge 2: Eternal's FTORB uses two methods, *get_state()* and *set_state()*, for checkpointing and recovery. When a new replica is added to a group, FTORB invokes the *get_state()* method of an existing replica to obtain the state of that replica. It then invokes the *set_state()* method of the new replica to initialize its state and replays the messages from the message log. The new replica begins processing requests, from that point onwards in the message sequence.

The Load Balancer is a stateful server and, thus, the challenge is to identify the state of the Load Balancer and to implement the *get_state()* and *set_state()* methods for it.

Solution: We identified the state of the Load Balancer, in particular the state that was stored in its table when the peer servers registered with the Load Balancer and that associate the object ids of the peer servers with their object group references. The Load Balancer uses this state to identify different peer servers, select a peer server to process a client's request, and supply object group references to the clients.

We then coded the *get_state()* and *set_state()* methods for the Load Balancer, which retrieve the state from one replica of the Load Balancer and set it within another replica of the Load Balancer, respectively.

3.2 Challenges in Designing the Client and Server Applications

Next we discuss some of the challenges that we faced in designing the client and server applications.

Challenge 3: The *get_state()* and *set_state()* methods must be coded for the client and server applications to make them fault tolerant, as described above. The challenge is to identify the state of the client and server applications.

Solution: For each client and server application in our examples, we identified the state and coded the *get_state()* and *set_state()* methods for that client and server. Global variables and data structures that are retained from one request to the next must be recorded, but local variables, such as loop indices, need not be.

Challenge 4: Only one peer server per peer group invokes the *create_object_group()* method of the Load Balancer to create a peer group. Once the Load Balancer has created the peer group, the other peer servers first obtain the object group reference of the peer group and then invoke the *register_servant()* method to add themselves to the peer group. TAO's Load Balancer does not provide this functionally. The challenge is to incorporate it within our integrated infrastructure.

Solution: One possible solution to this challenge was to recode the method *create_object_group()* in the TAO Load Balancer. Another possible solution was to make the necessary changes within the server to obtain the object group reference of the peer group that was created.

Because *create_object_group()* is part of TAO's Load Balancer, we adopted the second solution. Thus, whenever we create a peer group, we write its object group reference to an IOR file. Other peer servers that wish to add themselves to the peer group read the object group reference from the IOR file.

4 Performance Measurements

We describe three of the experiments that we performed for the integrated infrastructure and give the corresponding performance measurements.

The experiments were performed on up to nine Pentium III 1 GHz computers, connected by a 100 Mbps Ethernet switch. The computers ran the Linux Red Hat 8.0 operating system and the TAO CORBA ORB [11].

The applications consisted of a client that invokes a server remotely across the network. The client incurs a random delay (think time) between requests. The server performs a nominal processing operation of 30000 microseconds before responding to the client. Request and reply messages are 1kByte each.

We measured the response time seen by a client (*i.e.*, the time interval in microseconds between a client's issuing a request and receiving a reply from the server) and the throughput of a peer server (*i.e.*, the number of requests per second handled by the peer server).

4.1 Experiment 1: Decrease in Response Time with Load Balancing

Fig. 6 shows the testbed for this experiment, which consists of the Load Balancer, three clients and three peer servers, each having one replica. Each of the objects ran on a different processor, with FTORB running on all of the processors. First, we performed the experiment with one peer server serving the three clients. Next, we performed the experiment with three peer servers serving the three clients.

The graph in Fig. 6 shows the two configuration setups on the horizontal axis and the response time for a client's request on the vertical axis. Point A represents the setup with one peer server serving the three clients, and point B represents the setup with three peer servers serving the three clients. The experimental results show that the response time improved by 32%, when the clients' requests are load balanced across the three peer servers.

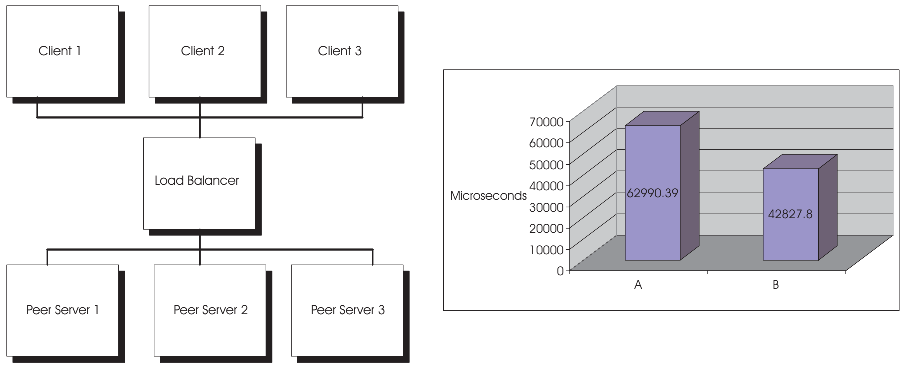

Fig. 6. Experiment 1. Testbed setup and response time for a client's request with load balancing: (A) one peer server and (B) three peer servers

4.2 Experiment 2: Increase in Throughput with Load Balancing

Fig. 7 shows the testbed for this experiment, which consisted of the Load Balancer, five clients and two peer servers, each having one replica. Each of the objects was hosted on a different computer, and FTORB was running on all of the computers.

The graph in Fig. 7 shows the server throughput curves without load balancing (bottom curve) and with load balancing (top curve). The vertical axis represents the throughput in requests per second. The horizontal axis represents decreasing random delays (think times) between requests at the clients. Initially, both of the throughput curves increase and there is no queuing at the peer server. After the peer server reaches its maximum processing capacity, the throughput curves flatten out and arriving requests from the clients are queued.

With load balancing across the two peer servers, we observed that the clients are distributed in groups of two and three between the two peer servers. This division resulted in higher throughput and decreased load on the single peer server without load balancing. When we decreased the number of clients communicating with the single server without load balancing from five to three, the throughput of the server increased by 15%. Thus, integrating load balancing resulted in a 15% increase in throughput.

4.3 Experiment 3: Increase in Response Time with Replication

Fig. 8 shows the testbed for this experiment, with the Load Balancer, one client and one peer server, having one, two or three replicas. Each of the objects ran on a different pro-

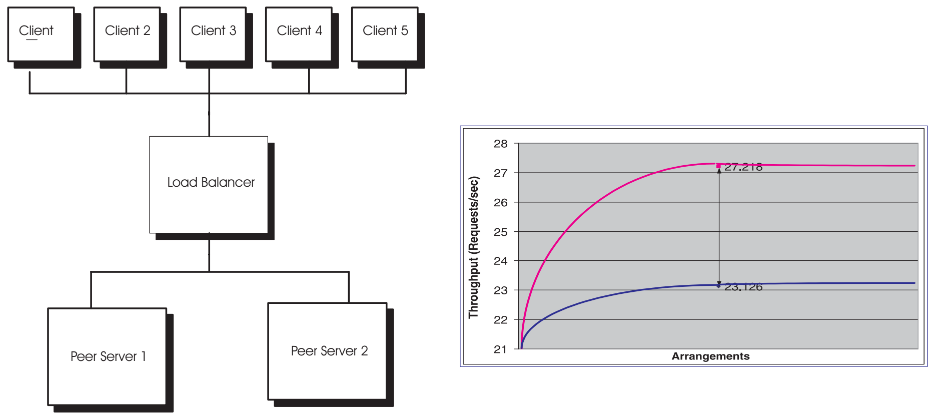

Fig. 7. Experiment 2. Testbed setup and throughput of a peer server: (A) bottom curve without load balancing and (B) top curve with load balancing

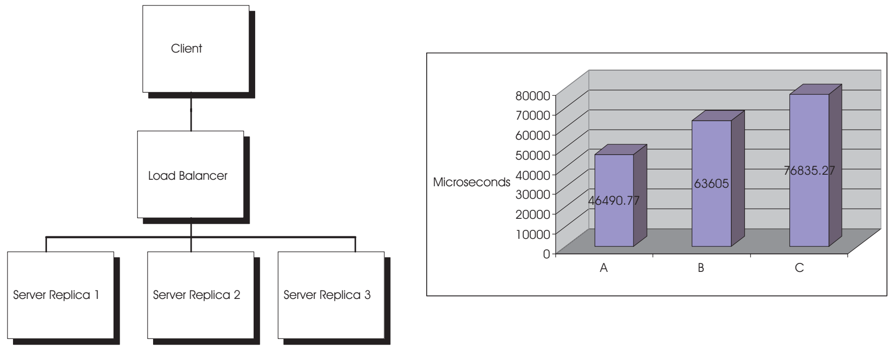

Fig. 8. Experiment 3. Testbed setup and response time: (A) one server replica, (B) two server replicas and (C) three server replicas

cessor, with FTORB running on all of the processors. In this experiment, we investigated the response time as the number of replicas of a peer server is increased.

The graph in Fig. 8 shows the results of the experiment. The vertical axis shows the response time in microseconds. Points A, B and C on the horizontal axis represent the three configurations when one, two and three server replicas are running. The results show that there is a 37% overhead when the number of replicas is increased from one to two and a 20% overhead when the number of replicas is increased from two to three. The primary cause of the increased overhead is the Totem multicast protocol, the latency of which increases linearly with the number of processors on the ring [12].

5 Related Work

Research on load balancing for distributed systems typically focuses on dynamic strategies and algorithms that maximize throughput and minimize overhead, and does not also deal with fault tolerance. Likewise, research on fault tolerance for distributed systems focuses primarily on strategies and mechanisms that achieve a high quality of fault tolerance, and does not also address load balancing or other quality of service issues. A few infrastructures have been developed that provide both fault tolerance and load balancing. Generally, they do not provide fault tolerance transparently but, rather, depend on modifications to the application programs.

Beowulf is a parallel processing environment that provides dynamic, on-board, adaptive distribution of processing tasks across a heterogeneous network of processors. It allocates processing to off-board resources as appropriate and as resources become available. In [2] Bennett, Davis and Kunau describe ParaSort, a distributed parallel data allocation sorting algorithm with automatic load balancing and fault tolerance that operates in the Beowulf environment. The fault tolerance is explicitly programmed into the application, in contrast to our more transparent fault tolerance approach.

AQuA [10] is a dependability framework that provides object replication and fault tolerance for CORBA applications. AQuA exploits the group communication facilities and message ordering guarantees of the Ensemble and Maestro toolkits to ensure replica consistency. AQuA supports both active and passive replication, with state transfer to synchronize the states of the backup and primary replicas for passive replication. AQuA also addresses resource management and other quality of services issues, but not load balancing based on the CORBA standard like our infrastructure does.

Ho and Leong [3] have extended the CORBA Event Service with load balancing and fault tolerance in a transparent manner. Their framework replicates event channels and shares the load among the replicas, using both static and dynamic load balancing, to improve scalability. It monitors the event channel replicas and, if an event channel replica becomes faulty, it transfers the consumers of the faulty event channel replica to another event channel replica and restarts the faulty event channel replica. Unlike our infrastructure which replicates stateful application objects, their framework replicates stateless event channels.

JBoss [4] is an open-source Java EJB/J2EE application server and, as such, it uses CORBA's IIOP protocol. JBoss has been extended with the JavaGroups group communication toolkit [1], a Java implementation of the Ensemble toolkit, to provide clustering, including load balancing, session state replication, and failover. JBoss uses an abstraction framework to isolate communication layers and, thus, like our infrastructure, achieves transparency to the applications and other middleware.

Petri, Bolz and Langendorfer [9] have developed a system that provides load balancing and fault tolerance for compute-intensive scientific applications. Their system uses a global virtual name space for groups of processes distributed across a workstation cluster. Applications use the same virtual names for operating system objects, independent of their location. System calls are interposed via the debugging interface, and parameters are translated between name spaces. Thus, like our infrastructure, their system uses library interpositioning to achieve transparency to the applications.

6 Conclusion

We have presented an integrated software infrastructure, based on Eternal's FTORB and TAO's Load Balancer, that renders distributed applications, based on CORBA, both fault tolerant and load balanced. We have discussed challenges that we faced in the integration and solutions to those challenges. We have also presented performance measurements, which show that the integrated infrastructure resulted in some overhead, as one would expect. However, the integrated infrastructure increases the robustness of the services that the applications provide to the clients and results in improved response time for the clients and throughput of the servers. Our work has shown that integrating middleware components, such as TAO's Load Balancer and Eternal's FTORB, that provide orthogonal non-functional properties might require modifications to one or both components, because each makes assumptions that the other might not satisfy.

References

1. B. Ban, "Design and implementation of a reliable group communication toolkit for Java," (September 1998), *http://www.cs.cornell.edu/home/bba/Coots.ps.gz*.
2. B. H. Bennett, E. Davis and T. Kunau, "Beowulf parallel processing for dynamic load balancing," *Proceedings of the IEEE Aerospace Conference,* vol. 4 (March 2000), Piscataway, NJ, pp. 389-395.
3. K. S. Ho and H. V. Leong, "An extended CORBA event service with support for load balancing and fault tolerance," *Proceedings of the IEEE International Symposium on Distributed Objects and Applications* (September 2000), Antwerp, Belgium, pp. 49-58.
4. B. Burke and S. Labourey, "Clustering with JBoss 3.0" (July 2002), *http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html*.
5. 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*, vol. 39, no. 4 (April 1996), pp. 54-63.
6. P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Strongly consistent replication and recovery of fault-tolerant CORBA applications," *Computer System Science and Engineering Journal*, vol. 17, no. 2 (March 2002), pp. 103-114.
7. Object Management Group, Fault Tolerant CORBA, OMG Technical Committee Document ptc/2000-04-04 (April 2000).
8. Object Management Group, CORBA Load Balancing and Monitoring Specification, OMG Document mars/02-10-14 (October 2002).
9. S. Petri, M. Bolz and H. Langendorfer, "Migration and rollback transparency for arbitrary distributed applications in workstation clusters," *Proceedings of the Parallel and Distributed Processing Symposium* (March-April 1998), Berlin, Germany, pp. 159-170.
10. Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M.Seri, "AQuA: An adaptive architecture that provides dependable distributed objects," *IEEE Transactions on Computers*, vol. 52, no. 1 (January 2003), pp. 31-50.
11. D. C. Schmidt, D. L. Levine and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4 (April 1998), pp. 294-324.
12. E. Thomopoulos, L. E. Moser and P. M. Melliar-Smith, "Latency analysis of the Totem single-ring protocol," *ACM/IEEE Transactions on Networking*, vol. 9, no. 5 (October 2001), pp. 669-680.
13. W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Unification of replication and transaction processing in three-tier architectures," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vienna, Austria (July 2002), pp. 290-297.