An Efficient Adaptive Load Balancing Service for CORBA

Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt
{ossama, coryan, schmidt}@uci.edu
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697-2625, USA [*]

February 13, 2001

A subset of this paper will appear in the Distributed Systems Engineering Journal's "Online" edition, March 2000.

Abstract

CORBA is increasingly popular as distributed object computing middleware for systems with stringent quality of service (QoS) requirements, including scalability and dependability. One way to improve the scalability and dependability of CORBA-based applications is to balance system processing load among multiple server hosts. Load balancing can help improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it can help improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures.

This paper presents three contributions to research on CORBA-based load balancing. First, we describe deficiencies with common load-balancing techniques, such as introducing unnecessary overhead or not adapting dynamically to changing load conditions. Second, we present a novel adaptive load balancing service that can be implemented efficiently using standard CORBA features. Finally, we present the results of benchmark experiments that evaluate the pros and cons of different load balancing strategies empirically by measuring the overhead of each strategy and showing how well each strategy balances system load.

Keywords: Middleware, patterns, CORBA, load balancing.

1 Introduction

Motivation: The growth of online Internet services during the past decade has increased the demand for scalable and dependable distributed computing systems. For example, e-commerce systems and online stock trading systems concurrently service many clients that transmit a large, often bursty, number of requests. To protect initial hardware investments and avoid overcommitting resources these systems scale incrementally by connecting servers via high-speed networks and either purchasing new servers as the number of clients increase or leasing server cycles during peak hours.

An increasingly popular and cost effective technique to improve networked server performance is *load balancing*, where hardware and/or software mechanisms determine which server will execute each client request. Load balancing mechanisms distribute client workload equitably among back-end servers to improve overall system responsiveness. These mechanisms can be provided in any or all of the following layers in a distributed system:

• **Network-based load balancing:** This type of load balancing is provided by IP routers and domain name servers (DNS) that service a pool of host machines. For example, when a client resolves a hostname, the DNS can assign a different IP address to each request dynamically based on current load conditions. The client then contacts the designated back-end server, unaware that a different server could be selected for its next DNS resolution. Routers can also be used to bind a TCP flow to any back-end server based on the current load conditions and then use that binding for the duration of the flow.

High volume Web sites often use network-based load balancing at the *network* layer (layer 3) and *transport* layer (layer 4). Layer 3 and 4 load balancing (referred to as "switching" in the trade literature [1]), use the IP address/hostname and port, respectively, to determine where to forward packets. Load balancing at these layers is somewhat limited, however, by the fact that they do not take into account the content of client requests. Instead, higher-layer mechanisms–such as the so-called layer 5 switching described below–perform load balancing in accordance with the content of requests, such as

[*]This work was funded in part by Automated Trading Desk, BBN, Cisco, DARPA contract 9701516, and Siemens MED.

pathname information within a URL.

• **OS-based load balancing:** This type of load balancing is provided by distributed operating systems via *clustering*, *load sharing*[1], and *process migration* [2] mechanisms. Clustering is a cost effective way to achieve high-availability and high-performance by combining many commodity computers to improve overall system processing power. Processes can then be distributed transparently among computers in the cluster.

Clusters generally employ load sharing and process migration. Balancing load across processors–or more generally across network nodes–can be achieved via *process migration* mechanisms [3], where the state of a process is transferred between nodes. Transferring process state requires significant platform infrastructure support to handle platform differences between nodes. It may also limit applicability to programming languages based on virtual machines, such as Java.

• **Middleware-based load balancing:** This type of load balancing is performed in middleware, often on a per-session or per-request basis. For example, layer 5 switching [1] has become a popular technique to determine which Web server should receive a client request for a particular URL. This strategy also allows the detection of "hot spots," *i.e.*, frequently accessed URLs, so that additional resources can be allocated to handle the large number of requests for such URLs.

This paper focuses on another type of middleware-based load balancing supported by *object request brokers* (ORBs), such as CORBA [4]. ORB middleware allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [5]. Moreover, ORBs can determine which client requests to route to which object replicas on which servers.

Middleware-based load balancing can be used in conjunction with the specialized network-based and OS-based load balancing mechanisms outlined above. It can also be applied on top of commodity-off-the-shelf (COTS) networks and operating systems, which helps reduce cost. In addition, middleware-based load balancing can provide semantically-rich customization hooks to perform load balancing based on a wide range of application-specific load balancing conditions, such as run-time I/O vs. CPU overhead conditions.

CORBA load balancing example: To illustrate the benefits of middleware-based load balancing, consider the CORBA-based online stock trading system shown in Figure 1. A distributed online stock trading system creates sessions through which trading is conducted. This system consists of multiple

[1]"Load sharing" should not be confused with "load balancing," *e.g.*, processing resources can be *shared* among processors but not necessarily *balanced*.

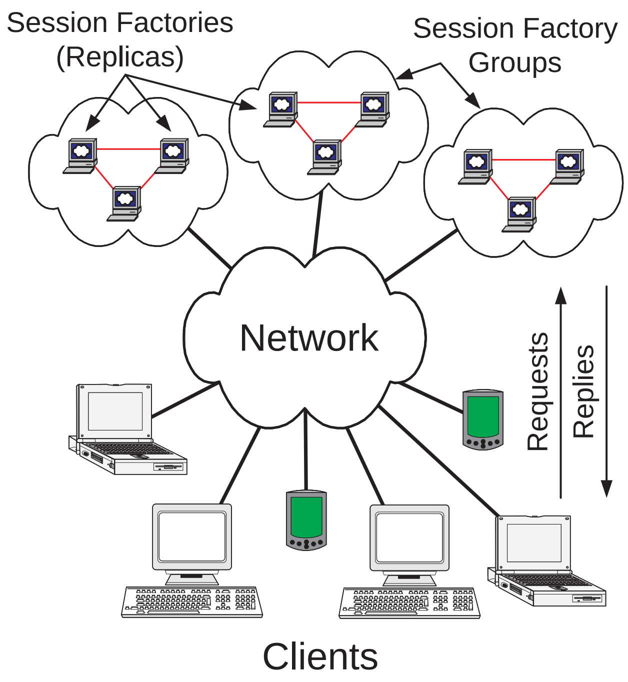

Figure 1: A Distributed Online Stock Trading System

back-end servers–called *replicas*–that process session creation requests sent by clients over a network. A replica is an object that can perform the same tasks as the original object. Server replicas that perform the same operations can be grouped together into *back-end server groups*, which are also known as *replica groups* or *object groups*.

For the example in Figure 1, a *session factory* [6] is replicated in an effort to reduce the load on any given factory. The load in this case is a combination of (1) the average number of session creation requests per unit time and (2) the total amount of resources employed currently to create sessions at a given location. Loads are then balanced across all replicas in the session factory replica group. The replicas need not reside at the same location.

The sole purpose of session factories is to create stock trading sessions. Therefore, factories need not retain state, *i.e.*, they are *stateless*. Moreover, in this type of system client requests arrive dynamically–not deterministically–and the duration of each request many not be known *a priori*.

These conditions require that the distributed online stock trading system be able to redistribute requests to replicas dynamically. Otherwise, one or more replicas may potentially become overloaded, whereas others will be underutilized. In other words, the system must *adapt* to changing load conditions. In theory, applying adaptivity in conjunction with multiple back-end servers can

- Increase the scalability and dependability of the system;

- Reduce the initial investment when the number of clients is small; and
- Allow the system to scale up gracefully to handle more clients and processing workload in larger configurations.

In practice, achieving this degree of scalability and dependability requires a sophisticated load balancing service. Ideally, this service should be transparent to existing online stock trading components. Moreover, if incoming requests arrive dynamically, a load balancing service may not benefit from *a priori* QoS specifications, scheduling, or admission control and must therefore adapt dynamically to changes in run-time conditions.

The CORBA load balancing service described in this paper fulfills the needs of applications with high scalability requirements, such as the online stock trading system described above. In contrast, neither the network-based nor OS-based load balancing solutions provide as straightforward, portable, and economical a means of adapting load balancing decisions based on application-level request characteristics, such as content and duration.

Paper organization: The remainder of this paper is organized as follows: Section 2 outlines the pros and cons of alternative load balancing architectures; Section 3 evaluates the performance of alternative load balancing strategies empirically; Section 4 compares our adaptive middleware-based load balancing service with related work; and Section 5 presents concluding remarks.

2 Overview of Alternative CORBA Load Balancing Strategies and Architectures

In this section we describe a variety of strategies and architectures for devising CORBA load balancing services.

2.1 Load Balancing Strategies

There are various strategies for designing CORBA load balancing services. These strategies can be classified along the following orthogonal dimensions:

Client binding granularity: A load balancer *binds* a client request to a replica each time a load balancing decision is made. Specifically, a client's requests are bound to the replica selected by the load balancer. Client binding mechanisms include GIOP LOCATION_FORWARD messages, modified standard CORBA services, or *ad hoc* proprietary interfaces. Regardless of the mechanism, client binding can be classified according to its granularity, as follows:

- *Per-session* – Client requests will continue to be forwarded to the same replica for the duration of a *session*[2], which is usually defined by the lifetime of the client [7].
- *Per-request* – Each client request will be forwarded to a potentially different replica, *i.e.*, bound to a replica each time a request is invoked.
- *On-demand* – Client requests can be re-bound to another replica whenever deemed necessary by the load balancer. This design forces a client to send its requests to a different replica than the one it is sending requests to currently.

Balancing policy: When designing a load balancing service it is important to select an appropriate algorithm that decides which replica will process each incoming request. For example, applications where all requests generate nearly identical amounts of load can use a simple round-robin algorithm, while applications where load generated by each request cannot be predicted in advance may require more advanced algorithms. In general, load balancing policies can be classified into the following categories:

- *Non-adaptive* – A load balancer can use non-adaptive policies, such as a simple round-robin algorithm or a randomization algorithm, to select which replica will handle a particular request.
- *Adaptive* – A load balancer can use adaptive policies that utilize run-time information, such as the amount of idle CPU available on each back-end server, to select the replica that will handle a particular request.

2.2 Load Balancing Architectures

By combining the strategies described above in various ways, it is possible to create the alternative load balancing architectures described below. In the ensuing discussion, we evaluate the pros and cons of these strategies *qualitatively*. Section 3 then evaluates these different strategies *quantitatively*.

Non-adaptive per-session architectures: One way to design a CORBA load balancer is make to the load balancer select the target replica when a client/server session is first established, *i.e.*, when a client obtains an object reference to a CORBA object–namely the replica–and connects to that object, as shown in Figure 2.

Note that the balancing policy in this architecture is *non-adaptive* since the client interacts with the same server to which it was directed originally, regardless of that server's load conditions. This architecture is suitable for load balancing

[2]In the context of CORBA, a *session* defines the period of time during which a client is connected to a given server for the purpose of invoking remote operations on objects in that server.

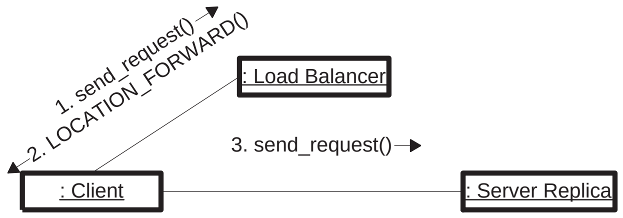

Figure 2: A Non-Adaptive Per-Session Architecture

policies that implement round-robin or randomized balancing algorithms.

Different clients can be directed to different object replicas by either using (1) a middleware activation daemon, such as a CORBA Implementation Repository [8] or (2) a lookup service, such as the CORBA Naming or Trading service. For example, Orbix [9] provides an extension to the CORBA Naming Service that returns references to object replicas in either a random or round-robin order.

Load balancing services based on a per-session client binding architecture can satisfy requirements for application transparency, increased system dependability, minimal overhead, and CORBA interoperability. The primary benefit of per-session client binding is that it incurs less run-time overhead than the alternative architectures described below.

Non-adaptive per-session architectures do not, however, satisfy the requirement to handle *dynamic* client operation request patterns adaptively. In particular, forwarding is performed only when the client binds to the object, *i.e.*, when it invokes its first request. Overall system performance may suffer, therefore, if multiple clients that impose high loads are bound to the same server, even if other servers are less loaded. Unfortunately, non-adaptive per-session architectures have no provisions to reassign their clients to available servers.

Non-adaptive per-request architectures: A non-adaptive per-request architecture shares many characteristics with the non-adaptive per-session architecture. The primary difference is that a client is bound to a replica *each time* a request is invoked in the non-adaptive per-request architecture, rather than *just once* during the initial request binding. This architecture has the disadvantage of degrading performance due to increased communication overhead, as shown in Section 3.2.

Non-adaptive on-demand architectures: Non-adaptive on-demand architectures have the same characteristics as their per-session counterparts described above. However, non-adaptive on-demand architectures allow re-shuffling of client bindings at an arbitrary point in time. Note that run-time information, such as CPU load, is not used to decide when to rebind clients. Instead, clients could be re-bound at regular time intervals, for example.

Adaptive per-session architecture: This architecture is similar to the non-adaptive per-session approach. The primary difference is that an adaptive per-session can use run-time load information to select the replica, thereby alleviating the need to bind new clients to heavily loaded replicas. This strategy only represents a slight improvement, however, since the load generated by clients can change after binding decisions are made. In this situation, the adaptive on-demand architecture offers a clear advantage since it can respond to dynamic changes in client load.

Adaptive per-request architectures: A more adaptive request architecture for CORBA load balancing is shown in Figure 3. This design introduces a front-end server, which is a

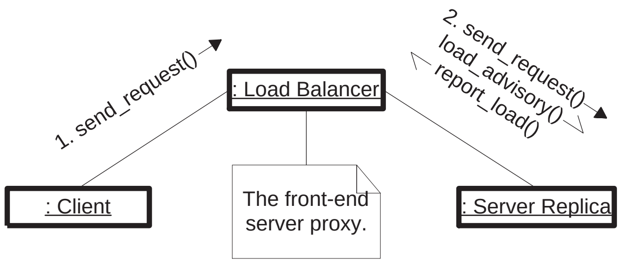

Figure 3: An Adaptive Per-request Architecture

proxy [10] that receives all client requests. In this case, the “front-end server” is the load balancer. The load balancer selects an appropriate back-end server replica in accordance with its load balancing policy and forwards the request to that replica. The front-end server proxy waits for the replica’s reply to arrive and then returns it to the client. Informational messages–called *load advisories*–are sent from the load balancer to replicas when attempting to balance loads. These advisories cause the replicas to either accept requests or redirect them back to the load balancer.

The primary benefit of an adaptive request forwarding architecture is its potential for greater scalability and fairness. For example, the front-end server proxy can examine the current load on each replica before selecting the target of each request, which may allow it to distribute load more equitably. Hence, this forwarding architecture is suitable for use with adaptive load balancing policies.

Unfortunately, this architecture can also introduce excessive latency and network overhead because each request is processed by a front-end server. Moreover, two new network messages are introduced:

1. The request from the front-end server to the replica; and
2. The corresponding reply from the back-end server (replica) to the front-end server.

In addition, to ensure that the system is scalable and dependable (*e.g.*, no single point of failure), multiple intermediate servers may be required. This configuration in turn requires complex algorithms that propagate the current load information to all front-end servers. It also requires a mechanism to assign clients to the correct front-end server. In a sense, therefore, the load balancing problem must be solved both for back-end *and* front-end servers, which complicates system design and implementation.

Adaptive on-demand architecture: This architecture is the primary focus of the remainder of this paper. As shown in Figure 4, clients receive an object reference to the load balancer

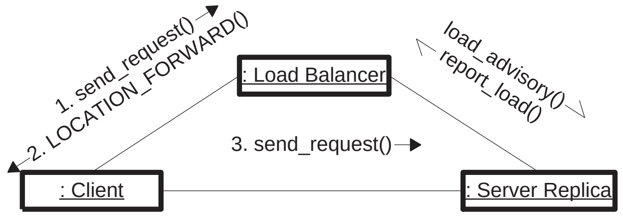

Figure 4: An Adaptive On-Demand Architecture

initially. Using CORBA's standard LOCATION_FORWARD mechanism, the load balancer can redirect the initial client request to the appropriate target server replica. CORBA clients will continue to use the new object reference obtained as part of the LOCATION_FORWARD message to communicate with this replica directly until they are redirected again or finish their conversation.

Unlike the non-adaptive architectures described earlier, adaptive load balancers that forward requests on-demand can monitor replica load continuously. Using this load information and the policies specified by an application, a load balancer can determine how equitably the load is distributed. When load becomes unbalanced, the load balancer can communicate with one or more replicas and request them to use the standard CORBA LOCATION_FORWARD mechanism to redirect subsequent clients back to the load balancer. The load balancer will then redirect the client to a less loaded replica. Upon receipt of a LOCATION_FORWARD message, a standard CORBA client ORB re-contacts the load balancer, which then redirects the client transparently to a less heavily loaded replica.

Using this architecture, the overall distributed object computing system can (1) recover from unequitable client/replica bindings while (2) amortizing the additional network and processing overhead over multiple requests. This strategy requires minimal changes to the application initialization code and no changes to the object implementations (servants) themselves.

The primary drawback with adaptive on-demand architectures is that server replicas must be prepared to receive messages from a load balancer and redirect clients to that load balancer. Although the required changes do not affect application logic, application developers must modify a server's initialization and activation components to respond to the load advisory messages mentioned above.

It is possible to overcome some drawbacks of adaptive on-demand load balancers, however, by applying standard CORBA portable interceptors [11]. Likewise, implementations based on the patterns [12] in the CORBA Component Model (CCM) [13] can implement load balancing without requiring changes to application code. In the CCM, a *container* is responsible for configuring the portable object adapter (POA) [5] that manages a component. Thus, TAO's adaptive on-demand load balancer just requires enhancing standard CCM containers so they support load balancing, without incurring other changes to application code.

3 Performance Results

For load balancing to improve the overall performance of CORBA-based systems significantly, the load balancing service must incur minimal overhead. A key contribution of TAO's load balancing service is that it increases overall system throughput by distributing requests across multiple back-end servers (replicas) without increasing round-trip latency and jitter significantly.

This section describes the design and results of several experiments we performed to measure the benefits of TAO's load balancing strategy empirically, as well as to demonstrate the limitations with the alternative load balancing strategies outlined in Section 2. The first set of experiments in Section 3.2 show the amount of overhead incurred by the request forwarding architectures described in this paper. The second set of experiments in Section 3.3 demonstrate how TAO's load balancer can maintain balanced loads dynamically *and* efficiently, whereas alternative load balancing strategies cannot.

3.1 Hardware/Software Benchmarking Platform

Benchmarks performed for this paper were run using three 733 MHz dual CPU Intel Pentium III workstations, and one 400 MHz quad CPU Intel Pentium II Xeon workstation, all running Debian GNU/Linux "potato" (GLIBC 2.1), with Linux kernel version 2.2.16. GNU/Linux is an open-source operating system that supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. All workstations are connected through a 100 Mbps ethernet switch. This testbed is depicted in Figure 5. All benchmarks were run in the POSIX

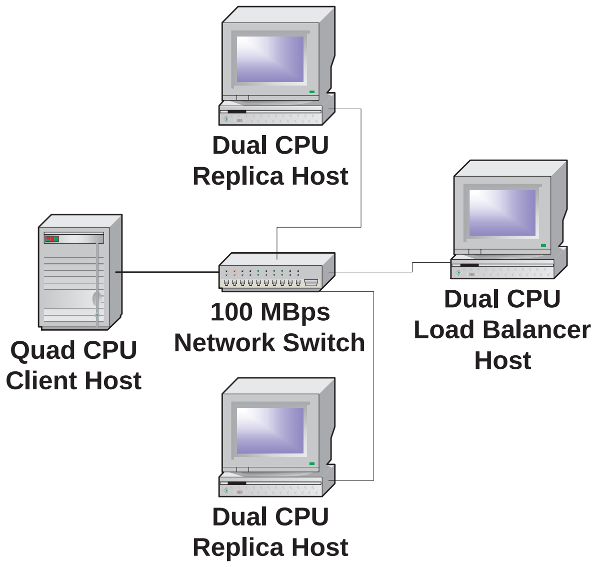

Figure 5: Load Balancing Experiment Testbed

real-time thread scheduling class [14]. This scheduling class enhances the integrity of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

The core CORBA benchmarking software is based on the "`Latency`" performance test distributed with the TAO open-source software release.[3] Figure 1 illustrates the basic design of this performance test. All benchmarks use one of the following variations of the `Latency` test:

1. Classic Latency test: In this benchmark, we use high-resolution OS timers to measure the throughput, latency, and jitter of requests made on an instance of a CORBA object that verifies a given integer is prime. Prime number factorization provides a suitable workload for our load balancing tests since each operation runs for a relatively long time. In addition, it is a stateless service that shields the results from transitional effects that would otherwise occur when transferring state between load balanced stateful replicas.

2. Latency test with non-adaptive per-request load balancing strategy: This variant of `Latency` test was designed to demonstrate the performance and scalability of *optimal* load balancing using per-request forwarding as the underlying request forwarding architecture. This variant added a specialized "forwarding server" to the test, whose sole purpose was to forward requests to a target server at the fastest possible rate. No changes were made to the client.

3. Latency test with TAO's adaptive on-demand load balancing strategy: This variant of the `Latency` test added support for TAO's adaptive on-demand load balancer to the classic `Latency` test. The `Latency` test client code remained unchanged, thereby preserving client transparency. This variant quantified the performance and scalability impact of TAO's adaptive on-demand load balancer.

[3]See `$TAO_ROOT/performance-tests/Latency/` in the TAO release for the source code of this benchmark.

3.2 Benchmarking the Overhead of Load Balancing Mechanisms

These benchmarks measure the degree of end-to-end overhead incurred by adding load balancing to CORBA applications.

Overhead measurement technique: The overhead experiments presented in this paper compute the throughput, latency, and jitter incurred to communicate between a single-threaded client and a single-threaded server (*i.e.*, one replica) using the following four request forwarding architectures:

1. No load balancing: To establish a performance baseline without load balancing, the `Latency` performance test was first run between a single-threaded client and a single-threaded server (one replica) residing on separate workstations. These results reflect the baseline performance of a TAO client/server application.

2. A non-adaptive per-session client binding architecture: We then configured TAO's load balancer to use the non-adaptive per-session load balancing strategy when balancing loads on a `Latency` test server. We did this by simply adding the registration code to the `Latency` test server implementation, which causes the replica to register itself with the load balancer so that it can be load balanced. No changes to the core `Latency` test implementation were made. Since the replica sends no feedback to the load balancer, this benchmark establishes a baseline for the best performance achievable by a load balancer that utilizes a per-session client binding granularity.

3. A non-adaptive per-request client binding architecture: Next, we added a specialized non-adaptive per-request "forwarding server" to the original `Latency` test. This server just forwards client requests to an unmodified backend server. The forwarding server resided on a different machine than either the client or backend server, which themselves each ran on separate workstations. Since the forwarding server is essentially a lightweight load balancer, this benchmark provides a baseline for the best performance achievable by a load balancer using a per-request client binding granularity.

4. An adaptive on-demand client binding architecture: Finally, TAO's adaptive on-demand client binding granularity was included in the experiment, which reacts to the current load on the `Latency` test server. TAO's load balancer, the client, and the server each ran on separate workstations, *i.e.*, three workstations were involved in this benchmark. No changes were made to the client portion of the `Latency` test, nor were any substantial changes made to the core servant implementation.

Overhead benchmark results: The results illustrated in Figure 6 quantify the latency imposed by adding load balancing–specifically request forwarding–to the `Latency` performance test. All overhead benchmarks were run with 200,000 iterations. As shown in this figure, a non-adaptive per-session approach imposes essentially no latency overhead to the classic `Latency` test. In contrast, the non-adaptive per-request approach more than doubles the average latency. TAO's adaptive on-demand approach adds little latency. The slight increase in latency incurred by TAO's approach is caused by

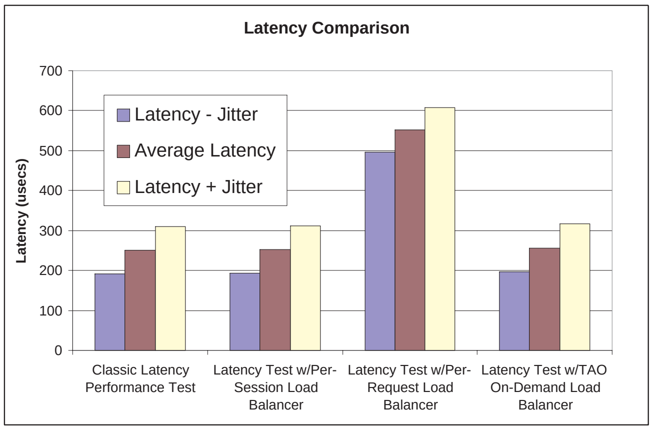

Figure 6: Load Balancing Latency Overhead

- The additional processing resources the load monitor needs to perform load monitoring; and
- The resources used when sending periodic load reports to the load balancer, *i.e.*, "push-based" load monitoring.

These results clearly show that it is possible to minimize latency overhead, yet still provide adaptive load balancing. As shown in Figure 6, the jitter did not change appreciably between each of the test cases, which illustrates that load balancing hardly affects the time required for client requests to complete.

Figure 7 shows how the average throughput differs between each load balancing strategy. Again, only one client and one server were used for this experiment. Not surprisingly, the throughput remained basically unchanged for the non-adaptive per-session approach since only one out of 200,000 requests was forwarded. The remaining requests were all sent to directly to the server, *i.e.*, all requests were running at their maximum speed.

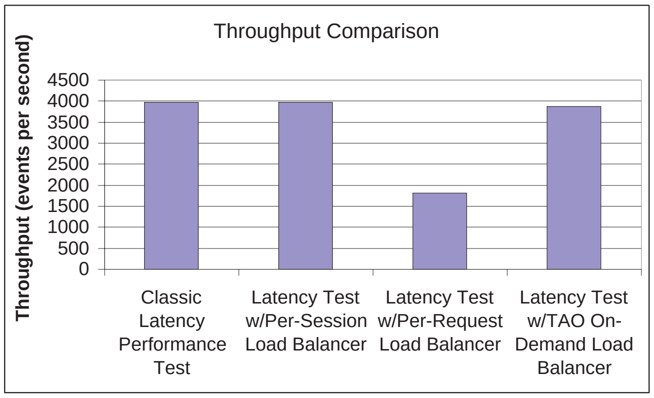

Figure 7: Load Balancing Throughput Overhead

Figure 7 illustrates that throughput decreases dramatically in the per-request strategy due to the fact that it (1) forwards requests on behalf of the client and (2) forwards replies received from the replica to the client, thereby doubling the communication required to complete a request. This architecture is clearly not suitable for throughput-sensitive applications.

In contrast, the throughput in TAO's load balancing approach only decreased slightly with respect to the case where no load balancing was performed. The slight decrease in throughput can be attributed to the same factors that caused the slight in increase in latency described above, *i.e.*, (1) additional resources used by the load monitor and (2) the communication between the load balancer and the load monitor.

3.3 Load Balancing Strategy Effectiveness

The following set of benchmarks quantify how effective each load balancing strategy is at maintaining balanced load across a given set of replicas. First, the effectiveness of the non-adaptive per-session load balancing strategy is shown. Next, the effectiveness of the adaptive on-demand strategy employed by TAO is illustrated. In all cases, we used the `Latency` test from the overhead benchmarks in Section 3.2 for the experiments.

Effectiveness measurement technique: The goal of this benchmark was to overload certain replicas in a group and then measure how different load balancing strategies handled the imbalanced loads. We hypothesized that loads across replicas should remain imbalanced when using non-adaptive per-session load balancing strategies. Conversely, when using adaptive load balancing strategies, such as TAO's adaptive load balancing strategy, loads across replicas should be balanced shortly after imbalances are detected.

To create this situation, four `Latency` test server replicas–each with a dedicated CPU–were registered with TAO's load balancer during each effectiveness experiment. Eight `Latency` test clients were then launched. Half the clients issued requests at a higher rate than the other half. For example, the first client issued requests at a rate of ten requests per-second, the second client issued requests at a rate of five requests per-second, the third at ten requests per-second, etc. The actual load was not important for this set of experiments. Instead, it was the *relative* load on each replica that was important, *i.e.*, a well balanced set of replicas should have relatively similar loads, regardless of the actual values of the load.

Effectiveness benchmark results: The results of the effectiveness tests are described below.

• **Non-adaptive per-session load balancing effectiveness:** For this experiment, TAO's load balancer was configured to use its *round-robin* load balancing strategy. This strategy does not perform any analysis on reported loads, but simply forwards client requests to a given replica. The client then continues to issue requests to the same replica over the lifetime of that replica. The load balancer thus applies the *non-adaptive per-session* strategy, *i.e.*, it is only involved during the initial client request.

Figure 8 illustrates the loads incurred on each of the `Latency` server replicas using non-adaptive per-session load

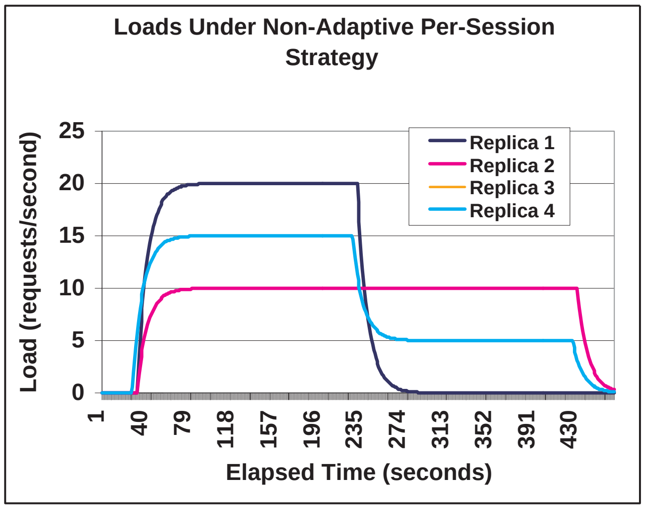

Figure 8: Effectiveness of Non-Adaptive Per-Session Load Balancing

balancing. The results quantify the degree to which loads across replicas become unbalanced by using this strategy. Since there is no feedback loop between the replicas and the load balancer, it is not possible to shift load from highly loaded replicas to less heavily loaded replicas.

Note that two of the replicas (3 and 4) had the same load. The line representing the load on replica 3 is obscured by the line representing the load on replica 4. In addition, note that the same number of iterations were issued by each client. Since some clients issued requests at a faster rate (10 Hz), however, those clients completed their execution before the clients with the lower request rates (5 Hz). This difference in request rate accounts for the sudden drop in load half way before the slower (*i.e.*, low load) clients completed their execution.

• **TAO's adaptive load balancing strategy effectiveness:** This test demonstrated the benefits of an adaptive load balancing strategy. Therefore, we increased the load imposed by each client and increased the number of iterations from 200,000 to 750,000. Four clients running at 100 Hz and another four running at 50 Hz were started and ended simultaneously.

Client request rates were increased to exaggerate load imbalance and to make the load balancing more obvious as it progresses. It was necessary to increase the number of iterations in this experiment because of the higher client request rates. If the number of iterations were capped at the 200,000 used in the overhead experiments in Section 3.2 this experiment could have ended before loads across the replicas were balanced.

As Figure 9 illustrates, the loads across all four replicas fluc-

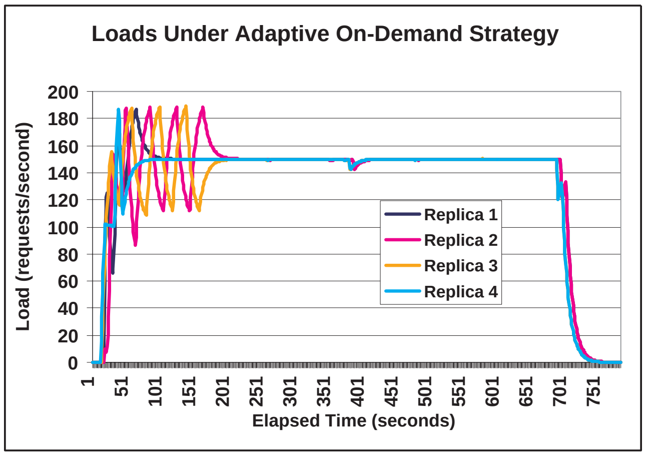

Figure 9: Effectiveness of Adaptive On-Demand Load Balancing

tuated for a short period of time until an equilibrium load of 150 Hz was reached.[4] The initial load fluctuations result from the load balancer periodically rebinding clients to less loaded replicas. By the time a given rebind completed, the replica load had become imbalanced, at which point the client was rebound to another replica. These initial fluctuations are typical of the adaptive load balancing hazards.

[4]The 150 Hz equilibrium load corresponds to one 100 Hz client and one 50 Hz client on each of the four replicas.

The load balancer required several iterations to balance the loads across the replicas, *i.e.*, to stabilize. Had it not been for the dampening built into TAO's adaptive on-demand load balancing strategy, it is likely that replica loads would have oscillated for the duration of the experiment. Dampening prevents the load balancer from basing its decisions on instantaneous replica loads, and to use average loads instead.

It is instructive to compare the results in Figure 9 to the non-adaptive per-session load balancing architecture results in Figure 8. Loads in the non-adaptive approach remained imbalanced. Using the adaptive on-demand approach, the overhead is minimized *and* loads remain balanced.

After it was obvious that the loads were balanced, *i.e.*, equilibrium was reached, the experiment was terminated. This accounts for the uniform drops in load depicted in Figure 9. Contrast this to the non-uniform drops in load that occured in the overhead experiments in Section 3.2, where clients were allowed to complete all iterations. In both cases, the number of iterations is less important than the fact that the iterations were executed to (1) illustrate the effects of load balancing and (2) ensure that the overall results were not subject to transient effects, such as periodic execution of operating system tasks.

The actual time required to reach the equilibrium load depends greatly on the load balancing strategy. The example above was based on a *minimum dispersion* strategy, which ensures that load differences fall within a certain tolerance, *i.e.*, it attempts to ensure that the average difference in load between each replica is minimized. A more sophisticated adaptive load balancing strategy could have been employed to improve the time to reach equilibrium. Regardless of the complexity of the adaptive load balancing strategy, these results show that adaptive load balancing strategies can maintain balanced loads across a given set of replicas.

4 Related Work

This section outlines related research on load balancing and describes how it compares and contrasts to our work on TAO's load balancing service. We first compare our middleware-based load balancing strategies with work at other levels of abstractions. Next, we describe how our work compares with other research on CORBA-based load balancing.

4.1 Related Research on Load Balancing

As discussed in Section 1, load balancing mechanisms have been implemented at various levels, such as in the network, OS, and middleware. Some implementations, such as the Condor [15] and Beowulf [16] clustering systems combine aspects from multiple levels. This paper focuses primarily on middleware-based load balancing, but many concepts and patterns used in middleware-based load balancing also apply to network-based and OS-based load balancing, as described below.

Network-based load balancing: Network-based load balancing implementations often make decisions based on the frequency with which a given location is accessed. The decision of where to service a request can be made at various stages along the path to its destination. For example, a router or DNS server could decide where to send a request.

Network-based load balancing has the disadvantage that load balancing decisions are based solely on the request target, which hampers flexibility greatly. However, recent developments in network-based load balancing do take advantage of request content. These hybrid implementations [1] provide finer-grained load analysis, which can improve load balancing decisions. Nevertheless, the choice of metric used in load balancing decisions is still restricted to the frequency with which a given target is accessed.

Unfortunately, frequency alone is not always an adequate load metric since some requests may incur large loads on the target host, *e.g.*, when Web servers process CGI requests. When combined with load balancing decisions based solely on target access frequency, the increased loads from such requests can degrade overall system performance. It is possible to analyze the content of each request to determine if it is a "high load" request, but this requires *a priori* knowledge of the requests behavior, which may not be feasible in many distributed computing systems.

OS-based load balancing: Some distributed operating systems, such as Chorus [17], can distribute processes transparently across remote OS endsystem nodes. Tools, such as GNU Queue [18], run a service that allows a user to run remote processes as if they were run on the local machine, *i.e.*, essentially transparently. Load balancing performed at this level has the advantage that it can be implemented transparently to applications. When loads are too high at their current location, running applications can be migrated to other nodes relatively transparently. As with the network-based architecture, however, load balancing at this level makes it hard to choose which metric to use when deciding where to move processes since application-level metrics and policies are not available at this level.

Middleware-based load balancing: Middleware-based load balancing implementations reside between the application and the OS/network. Middleware shields the application from tedious and error-prone low-level OS complexities, while also providing a powerful interface to make load balancing at this level as transparent as possible, if not completely transparent. Moreover, middleware can be implemented with sufficient flexibility to overcome the disadvantages in network-based and OS-based load balancing architectures.

CORBA is a prime example of a technology that provides the following capabilities needed to implement an effective load balancing service:

- Application developers can customize how their system is load balanced without being restricted by the limited– and often hard-coded–metrics available in network-based and OS-based load balancing.
- Applications can select at run-time the metric(s) used in load balancing decisions.
- New metrics can also be defined with relative ease by separating interface from implementation, *i.e.*, exposing a consistent interface for each implementation. For example, a load monitoring component can be implemented for a specific load metric, yet keep the load balancing service load metric neutral.

Moreover, a middleware-based load balancing service can be used in conjunction with network- and OS-based load balancing facilities, which supports some interesting load balancing combinations. For example, if an application just needs to balance load based on request frequency a middleware-based load balancer can delegate load balancing tasks to the network or OS layers. Conversely, the middleware-based load balancer itself could be load balanced at the network or OS level, thereby providing additional network/host resources for use by the middleware-based load balancer and other applications.

Other examples of middleware-based load balancing include some Web server implementations. Web servers can forward HTTP requests [19] to any one of a number hosts that replicate the target web page. Overall throughput can be increased using any of the load balancing approaches detailed in this paper. The key is that the web server performs the HTTP request load balancing.

The CORBA-based load balancing concepts detailed in this paper are generally applicable to other middleware implementations, such as COM+ [20]. In fact, a middleware-based load balancing service called *Component Load Balancing* (CLB) [21] is available from Microsoft for COM+ applications.

4.2 Related Work on CORBA-based Load Balancing

CORBA load balancing can be implemented at several levels in the OMG reference architecture, such as the following:

ORB-level: Load balancing can be implemented inside the ORB itself. For example, a load balancing implementation can take direct advantage of request invocation information available within the POA when it makes load balancing decisions. Moreover, middleware resources used by each object can also be monitored directly via this design, as described in [22]. For example, Inprise's VisiBroker implements a similar strategy, where Visibroker's object adapter [23] creates object references that point to Visibroker's Implementation Repository, called the OSAgent, that plays both the role of an activation daemon and a load balancer.

ORB-level techniques have the advantage that the amount of indirection involved when balancing loads can be reduced because load balancing mechanisms are closely coupled with the ORB *e.g.*, the length of communication paths is shortened. However, ORB-level load balancing has the disadvantage that it requires modifications to the ORB itself. Unless or until such modifications are adopted by the OMG, they will be proprietary, which reduces their portability and interoperability. Therefore, TAO's load balancing service does not rely on ORB-level extensions or non-standard features.

TAO's load balancing service does not require any modifications to the ORB core or object adapter. Instead, it takes advantage of standard mechanisms in CORBA 2.X to implement adaptive load balancing. Like the Visibroker implementation and the strategies described in [22], TAO's approach is transparent to clients. Unlike the ORB-based approaches, however, our implementation only uses standard CORBA features. Thus, it can be ported to any C++ CORBA ORB that implements the CORBA 2.2 or newer specification.

Service-level: Load balancing can also be implemented as a CORBA service. For example, the research reported in [24] extends the CORBA Event Service to support both load balancing and fault tolerance. Their system builds a hierarchy of *event channels* that fan out from event source *suppliers* to the event sink *consumers*. Each event consumer is assigned to a different leaf in the event channel hierarchy, and both fixed and adaptive load balancing is performed to distribute consumers evenly. In contrast, TAO's load balancing service can be used for application defined objects, as well as event services.

Various commercial CORBA implementations also provide service-level load balancing. For example, IONA's Orbix [25] can perform load balancing using the CORBA Naming Service. Different replicas are returned to different clients when they resolve an object. This design represents a typical non-adaptive per-session load balancer, which suffers from the disadvantages described in Section 2. BEA's WebLogic [26] uses a per-request load balancing strategy, also described in Section 2. In contrast, TAO's load balancing service does not incur the per-request network overhead of the BEA strategy, yet can still adapt to dynamic changes in the load, unlike Orbix's load balancing service.

5 Concluding Remarks

As network-centric computing becomes more pervasive and applications become more distributed, the demand for greater scalability and dependability is increasing. Distributed system scalability can degrade significantly, however, when servers become overloaded by the volume of client requests. To alleviate such bottlenecks, load balancing mechanisms can be used to distribute system load across object replicas residing on multiple servers.

Load can be balanced at several levels, including the network, OS, and middleware. Network-based and OS-based load balancing architectures suffer from several limitations:

- The lack of flexibility arises from the inability to support *application-defined* metrics at run-time when making load balancing decisions.
- The lack of adaptability occurs due to the absence of load-related feedback from a given set of replicas, as well as the inability to control if and when a given replica should accept additional requests.

Thus, middleware-based load balancing architectures–particularly those based on standard CORBA–have been devised to overcome the limitations with network-based and OS-based load balancing mechanisms outlined above.

This paper illustrates the performance of adaptive middleware-based load balancing mechanisms developed using the standard CORBA features provided by the TAO ORB [27]. Though CORBA provides solutions for many distributed system challenges, such as predictability, security, transactions, and fault tolerance, it still lacks standard solutions to tackle other important challenges faced by distributed systems architects and developers. Chief among those missing facilities are load balancing, state caching, and state replication.

The CORBA-based load balancing service provided by TAO fills part of this gap by allowing distributed applications to be load balanced adaptively and efficiently. This service increases overall system throughput by distributing requests across multiple back-end server replicas without increasing round-trip latency substantially or assuming predictable, or homogeneous loads. As a result, developers can concentrate on their core application behavior, rather than wrestling with complex infrastructure mechanisms needed to make their application distributed, scalable, and dependable.

TAO and TAO's load balancing service have been applied to a wide range of distributed applications, including many telecommunication systems, aerospace/military systems, online trading systems, medical systems, and manufacturing process control systems. All the source code, examples, and documentation for TAO, its load balancing service, and its other CORBA services is freely available from URL `http://www.cs.wustl.edu/~schmidt/TAO.html`. A paper describing the design of TAO's load balancing service appears in [28].

References

[1] E. Johnson and ArrowPoint Communications, "A Comparative Analysis of Web Switching Architectures." http://www.arrowpoint.com/solutions/white_papers/ws_archv6.html, 1998.

[2] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design.* Harlow, England: Pearson Education Limited, 2001.

[3] F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," in *Proceedings of the* 7^{th} *International Conference on Distributed Computing Systems,* (Berlin, West Germany), pp. 18–25, IEEE, Sept. 1987.

[4] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* 2.4 ed., Oct. 2000.

[5] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++.* Addison-Wesley Longman, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[7] N. Pryce, "Abstract Session," in *Pattern Languages of Program Design* (B. Foote, N. Harrison, and H. Rohnert, eds.), Reading, MA: Addison-Wesley, 1999.

[8] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA,* vol. 41, Oct. 1998.

[9] S. Baker, *CORBA Distributed Objects using Orbix.* Addison Wesley Longman, 1997.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons, 1996.

[11] Adiron, LLC, *et al., Portable Interceptor Working Draft – Joint Revised Submission.* Object Management Group, OMG Document orbos/99-10-01 ed., October 1999.

[12] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2.* New York, NY: Wiley & Sons, 2000.

[13] BEA Systems, *et al., CORBA Component Model Joint Revised Submission.* Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

[14] Khanna, S., *et al.*, "Realtime Scheduling in SunOS 5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

[15] J. Basney and M. Livny, "Deploying a High Throughput Computing Cluster," *High Performance Cluster Computing,* vol. 1, May 1999.

[16] D. Ridge, D. Becker, P. Merkey, and T. Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," in *Proceedings, IEEE Aerospace*, IEEE, 1997.

[17] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser, "Overview of the CHORUS Distributed Operating Systems," Tech. Rep. CS-TR-90-25, Chorus Systems, 1990.

[18] W. G. Krebs, "Queue Load Balancing / Distributed Batch Processecing and Local RSH Replacement System." http://www.gnuqueue.org/home.html, 1998.

[19] Sun-Netscape Alliance, "Technical Overview of Netscape Application Server 4.0." http://www.iplanet.com/products/whitepaper/whitepaper_3.html, 2000.

[20] D. Box, *Essential COM*. Addison-Wesley, Reading, MA, 1997.

[21] T. Ewald, "Use Application Center or COM and MTS for Load Balancing Your Component Servers." http://www.microsoft.com/msj/0100/loadbal/loadbal.asp, 2000.

[22] M. Lindermeier, "Load Management for Distributed Object-Oriented Environments," in *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.

[23] I. Inprise Corporation, "VisiBroker for Java 4.0: Programmer's Guide: Using the POA." http://www.inprise.com/techpubs/books/vbj/vbj40/programmers-guide/poa.html, 1999.

[24] K. S. Ho and H. V. Leong, "An Extended CORBA Event Service with Support for Load Balancing and Fault-Tolerance," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, (Antwerp, Belgium), OMG, Sept. 2000.

[25] IONA Technologies, "Orbix 2000." www.iona-iportal.com/suite/orbix2000.htm.

[26] BEA Systems Inc., "WebLogic Administration Guide." http://edoc.bea.com/wle/.

[27] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[28] O. Othman, C. O'Ryan, and D. C. Schmidt, "The Design of an Adaptive CORBA Load Balancing Service," *IEEE Distributed Systems Online*, vol. 2, Apr. 2001.