

Performance Evaluation of an Adaptive Middleware Load Balancing and Monitoring Service

Ossama Othman, Jaiganesh Balasubramanian, and Douglas C. Schmidt
{ossama,jai,schmidt}@dre.vanderbilt.edu
Institute for Software and Integrated Systems
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203, USA

Abstract

As middleware-based distributed applications become more pervasive, the need to improve the scalability of these applications becomes increasingly important. One way to improve scalability is via load balancing. Earlier generations of middleware-based load balancing services were simplistic, however, since they focused on specific use-cases and environments, which made it hard to use these services for anything other than a small class of distributed applications. This lack of generality also often forced continuous redevelopment of application-specific load balancing services, which increases distributed applications deployment and optimization costs. Recent advances in the design and implementation of middleware-based load balancing services overcome these limitations through several techniques, including (1) support for adaptive load balancing strategies which allows a load balancer to be applied to a wide variety of applications, (2) load metric neutrality which further allows a load balancer to remain non-application specific, and (3) server-side transparency, which prevents application implementations from being complicated by adding load balancing support.

This paper presents the following contributions to research on adaptive middleware-based load balancing techniques: (1) it describes the design of Cygnus, which is an adaptive load balancing/monitoring service we have developed based on the CORBA middleware standard, (2) it presents the results of benchmarking experiments that systematically evaluate different load balancing strategies supported by Cygnus to indicate empirically how they improve scalability, and (3) illustrates when adaptive load balancing is more suitable than non-adaptive load balancing for use in middleware-based distributed applications.

1 Introduction

Load balancing is a technique that can be used to reduce contention on a given resource by distributing access among redundant instances of that resource. Distributed applications can employ load balancing in various ways and at various levels to improve overall system scalability. For example, heavily accessed Internet web sites often use load balancing at the network [1] and operating system [2] levels to improve performance and accessibility to certain resources, such as network hosts and host processes, respectively. Load balancing at these levels, however, may be unsuitable for certain types of distributed systems due to the lack of application-level control over load balancing policies, lack of extensible load metrics, and difficulty or inability of taking client request content into account when balancing loads.

For certain types of distributed systems, such as online stock trading, e-commerce, and total ship computing systems [3], middleware-based load balancing [4] can help improve scalability without incurring the limitations of load balancing at lower levels outlined above. Since middleware-based load balancing can be employed using various strategies and multiple metrics, determining the most suitable strategies for different types of distributed applications is hard without the guidance of systematic performance results. This paper focuses on the empirical evaluation of a middleware-based load balancing approach designed to be highly flexible and suitable for use by many distributed applications and systems.

As with load balancing done at other levels, middleware-based load balancing can be either *non-adaptive* or *adaptive*, depending on whether or not dynamic load conditions influence load balancing decisions. Adaptive load balancing is often the most flexible and desirable, however, since it can satisfy many distributed application requirements, such as (1) improved handling of erratic client request patterns, and (2) optimizing resource utilization under non-uniform loading con-

ditions. Although adaptive load balancing can be a powerful technology, there are challenges to applying it effectively, in particular determining the conditions under which the overhead of adaptive load balancing becomes prohibitive relative to less powerful non-adaptive strategies. This paper will empirically evaluate this issue an open-source¹ CORBA [5] implementation of middleware-based load balancing and monitoring service call *Cygnus* that we have developed to perform non-adaptive and adaptive load balancing.

Our earlier work on middleware-based load balancing has focused on (1) defining a nomenclature powerful enough to describe various forms of middleware load balancing [4], (2) creating a flexible, portable, optimized and low overhead load balancing model [6], (3) identifying key advanced load balancing service features that may be used to further improve or enhance the load balancing model mentioned above [7], and (4) determining how to implement a middleware-based load balancing service that provides efficient run-time performance using standard features found in middleware technologies. This paper explores a previously unexamined topic pertaining to middleware-based load balancing: *empirically evaluating the different strategies used in adaptive and non-adaptive middleware-based load balancing and determining when adaptive middleware-based load balancing is suitable for use by distributed applications.*

The remainder of this paper is organized as follows: Section 2 presents an overview of the *Cygnus* middleware-based load balancing and monitoring service that is used as the basis for the experiments conducted in this paper; Section 3 evaluates empirical results that demonstrate why adaptive middleware-based load balancing is a scalable solution for certain types of distributed systems; Section 4 compares and contrasts research that is related to our own work; and Section 5 presents concluding remarks and future work.

2 Overview of *Cygnus*

This section identifies the desirable properties of a middleware-based load balancing service and describes the key concepts and the design of the *Cygnus* load balancing and monitoring service, which was developed to satisfy these desired properties.

2.1 Desirable Properties

It is desirable to identify the salient properties of a load balancing service before we can discuss about the specific design of a load balancing service. Below are certain key properties

¹The source code for *Cygnus* is available from deuce.doc.wustl.edu/Download.html.

of a load balancing service, which we used as a motivation in designing *Cygnus*:

- **General purpose:** A load balancing service should make little or no assumptions about the types of applications whose loads it balances. The load balancing service should not be tied to a particular application and should be specific to a specific class of distributed systems.
- **Requires little or no application change:** Applications constantly evolve over time and new applications are also deployed when deemed necessary. Changing applications to include load balancing support will incur continuous re-development and deployment costs. This will also lead to development of non-optimized load balancing implementations, that are specific to particular class of applications. Hence load balancing service should be developed as a standalone entity to the application.
- **Transparent:** A load balancing service should balance loads in a transparent manner. A client invoking an operation on a server could have its request processed by a local server or a remote server. From the client's point of view, there should be no difference between local and remote execution and hence clients should not be aware of the load balancing being done.
- **Dynamic:** A load balancing service should allow application developers to choose between different load balancing strategies while adding load balancing support to their application. For example, applications whose requests all generate similar or uniform load can choose to use a simple round robin algorithm to select which group member will receive the request, while applications whose requests generate erratic loads need to use a much more advanced algorithm that utilizes run-time information such as CPU load, number of requests processed per unit time, etc, to select the group member that will handle the request. Based on these different application characteristics, load balancing strategies can be classified into adaptive and non-adaptive load balancing strategies. Hence it is imperative for a load balancing service to support both adaptive and non-adaptive load balancing strategies.
- **Scalable:** A load balancing service should assist in improving scalability of a distributed application by handling a large number of client requests and manage many servers in an efficient manner.
- **Extensible :** A load balancing service must provide mechanisms for application developers to develop their custom load balancing strategies and use it in their load

balancing/shedding decisions. The application developers should also be able to select different load metrics during runtime. Therefore the load balancing service has to support a wide range of load metrics.

2.2 Key Load Balancing Concepts

The key load balancing concepts and components used in this paper and Cygnus are defined below:

- **Load balancer**, which is a component that attempts to ensure application load is balanced across groups of servers. It is sometimes referred to as a “load balancing agent” or a “load balancing service.” A load balancer may consist of a single centralized server or multiple decentralized servers that collectively form a single logical load balancer.
- **Member**, which is a duplicate instance of a particular object on a server that is managed by a load balancer. It performs the same tasks as the original object. A member can either retain state (*i.e.*, be *stateful*) or retain no state at all (*i.e.*, be *stateless*).
- **Object group**, which is actually a group of *members* across which loads are balanced. Members in such groups implement the same remote operations.
- **Session**, which in the context of distribution middleware defines the period of time that a client is connected to a given server for the purpose of invoking remote operations on objects in that server.

Figure 1 illustrates the relationships between these components.

2.3 Cygnus Design Overview

The core architectural components found in Cygnus are depicted in Figure 2 and described below:

- **Load manager**, which is the application entry point for all load balancing tasks, *i.e.* it is a *mediator*.
- **Member locator**, which is the *interceptor* responsible for binding a client to a member.
- **Load analyzer**, which analyzes load conditions and triggers load shedding when necessary based on the load balancing *strategy* in use.
- **Load monitor**, which makes load reports available to the load manager.
- **Load alert**, which is a *mediator* through which load shedding is performed.

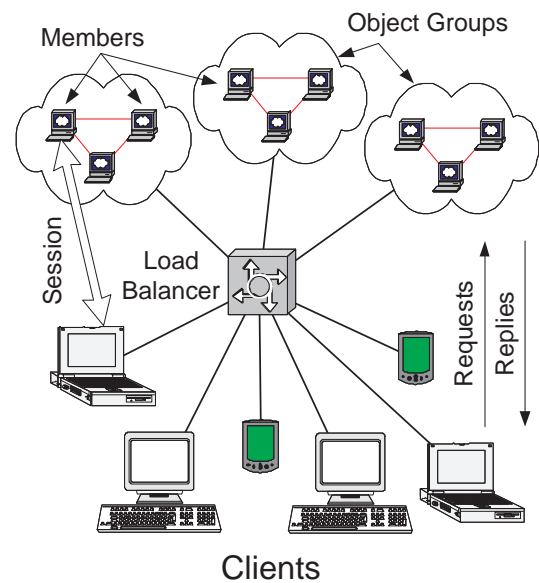


Figure 1: Key Load Balancing Concepts

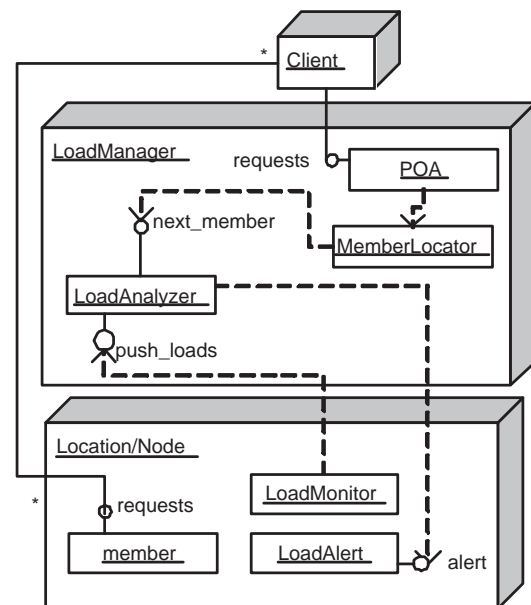


Figure 2: Components in the Cygnus LB/M Service

Design patterns [8] played a key role in the development of Cygnus. The following patterns were used in the design of Cygnus:

- **Interceptor pattern**, which enables load balancing to be added transparently without changing applications in use [9]. The member locator component implements the interceptor pattern to enable clients to resolve "next available" servers to process the request.
- **Strategy pattern**, which enables the load balancer to dynamically select the load metric to be used and use the load metric in its load balancing decisions [8].
- **Mediator pattern**, which allows loose coupling between the servers and load balancer [8]. The Load Monitor component acts as a mediator between the servers and the load balancer and defines interfaces which the load balancer can use to pull load reports from the load monitor.
- **Component Configurator pattern**, which allows dynamic (re)configurations of the load balancing service and add load monitor and the load analyzer components depending upon whether the application requires it or not [9]. In this way, the load balancer need not add components the application does not need.

A comprehensive discussion of the design of the Cygnus load balancing and monitoring service appears in [6].

3 Experimental Design and Empirical Results

To significantly improve the overall performance for a wide range of applications, a load balancing service should (1) incur minimum overhead, and (2) support both non-adaptive and adaptive load balancing strategies.

A key contribution of the Cygnus load balancing and monitoring (LB/M) service outlined in Section 2 is its ability to satisfy these requirements. It can increase overall system scalability of many types of middleware-based (and more specifically CORBA-based) distributed applications in an efficient, transparent and portable manner. The Cygnus LB/M service achieves scalability by distributing load across multiple backend servers, *i.e.* object group members (see Section 2.2) in a way that avoids significant overhead, round-trip latency and jitter.

The Cygnus LB/M service incurs minimal overhead by using per-session and on-demand client binding architectures described in detail in [10]. Cygnus provides the following built-in load balancing strategies (1) Round Robin (2) Random and

(3) Least Loaded. Random and Round Robin load balancing strategies are the common non-adaptive load balancing strategies while the Least Loaded load balancing strategy is a common adaptive load balancing strategy. Apart from this, Cygnus employs the use of the strategy pattern in the design of its load analyzer component, which allows the application developer to write a custom load balancing strategy to make load balancing/shedding decisions. A detailed discussion of the use of various patterns in the design of the Cygnus load balancing and monitoring service appears in [6].

This section describes the results of a set of experiments designed to empirically evaluate the performance of each of the built-in load balancing strategies provided by Cygnus, as well as to demonstrate when an adaptive load balancing strategy outperforms a non-adaptive load balancing strategy. Section 3.1 outlines the hardware and software platform used in these benchmarking experiments. Section 3.2 outlines the experiments used in these benchmarking measurements. Section 3.3 outlines the load metrics supported by the Cygnus load balancing/monitoring service and defines the load metric used in these benchmarking experiments. Section 3.4 outlines the runtime configuration for the Least Loaded load balancing strategy used in these benchmarking experiments. Section 3.5 presents the results from a set of experiments that illustrate the low overhead Cygnus load balancing/monitoring service adds to the distributed application. Section 3.6 presents the results from a set of experiments that illustrate the improved scalability attained by introducing Cygnus' adaptive and non-adaptive load balancing capabilities into a representative distributed application. Lastly, section 3.7 presents the results from a set of experiments that illustrate when the performance of adaptive load balancing strategy outperforms the performance of the non-adaptive load balancing strategy built in the Cygnus load balancing and monitoring service.

3.1 Hardware/Software Benchmarking Platform

Benchmarks performed for this paper were run on Emulab² using between 2 and 41 single CPU Intel Pentium III 850 MHz workstations, all running RedHat Linux 7.1. The Linux kernel is open-source and supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. All workstations were connected over a 100 Mbps LAN. This testbed is depicted in Figure 3. Benchmarks were all run in the POSIX real-time thread scheduling class [11] in order to enhance the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

²Emulab (www.emulab.net) is an NSF-sponsored testbed that facilitates simulation and emulation of different network topologies for use in experiments that require a large number of nodes.

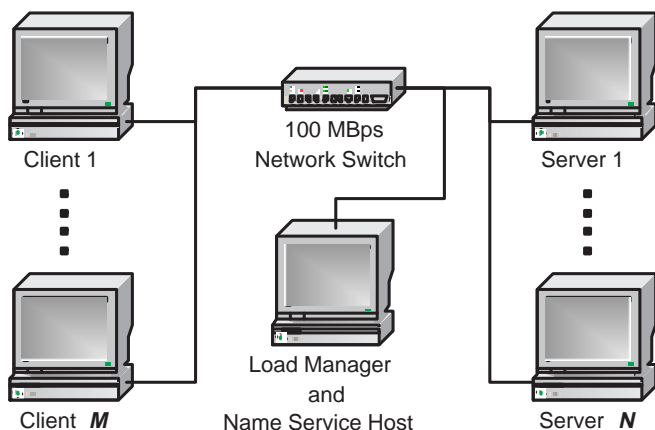


Figure 3: Load Balancing Experiment Testbed

3.2 Core Benchmarking Experiments

The core CORBA benchmarking software is based on the single-threaded form of the “Latency” performance test distributed with the TAO open-source software release.³ Only stateless objects are used as targets in this test. All benchmarks were configured to run at least 500,000 iterations. Furthermore, all benchmarks use one of the following variations of the Latency test:

1. **Latency test with Round Robin load balancing strategy.** In this benchmark, the Latency test was configured to employ the Round Robin load balancing strategy to improve scalability. This strategy is non-adaptive (*i.e.*, it does not consider dynamic load conditions) and simply chooses object group members to forward client requests to by cycling through the members in a given object group each time a new client attempts to invoke an operation on one of the group members. In other words, all the requests from the clients are equally distributed among the servers.
2. **Latency test with Random load balancing strategy.** In this benchmark, the Random load balancing strategy is used to improve scalability. It is a simple non-adaptive load balancing strategy that selects a member at random from the object group each time a new client attempts to invoke an operation on one of the group members.
3. **Latency test with Least Loaded load balancing strategy.** This final benchmark configuration uses Cygnus’ Least Loaded load balancing strategy to improve scalability. Unlike the Round Robin and Random tests, it uses an adaptive on-demand load balancing strategy. As its

³TAO/orbsvcs/performance-tests/LoadBalancing in the TAO release contains the source code for this benchmark.

name implies, it chooses the object group member with the lowest load, which is computed dynamically.

3.3 Load Metrics Supported

The Cygnus load balancing and monitoring (LB/M) service supports the following load metrics: (1) CPU load, and (2) requests-per-second. An application can be configured to use one of these load metrics. The load monitors are used to measure the loads at each resource based on the load metrics chosen. The core experiments in this benchmarking suite uses requests-per-second as the load metric. This metric calculates the average number of requests arriving per second at each server. For example, if the application chooses to use the Least Loaded load balancing strategy and the requests-per-second metric, load balancing and load shedding decisions are based on the average number of client requests arriving at a particular server at the particular instant.

3.4 Least Loaded Strategy Run-time Configuration

The Least Loaded load balancing strategy used for these benchmarking experiments was designed to exercise the adaptive load balancing support in Cygnus explicitly. In particular, the following configuration was used:

- A *load monitor* that measured the average number of requests arriving per second and residing within the server was registered with the Cygnus load balancing/monitoring service.
- A *reject threshold* of 25,000 events/second was set, which is the threshold at which Cygnus will avoid selecting a member with a load greater than or equal to that load.
- A *critical threshold* of 35,000 events/second was set, which is the threshold at which Cygnus informs servers to shed loads by redirecting requests back to Cygnus.
- A *dampening* value of 0.1 was set, which is the value that determines what fraction of a newly reported load is considered when making load balancing decisions.

The reject and critical threshold values were chosen in a manner so that the load experienced by the servers did not cause the load balancing service to alert the servers to shed load. This may cause the load balancing service to be contacted every now and then and this may incur serious overhead over the distributed application. Hence the clients may experience lesser throughput and increased roundtrip latency.

A dampening value of 0.1 was chosen so that the Cygnus load balancing/monitoring service takes into account a higher

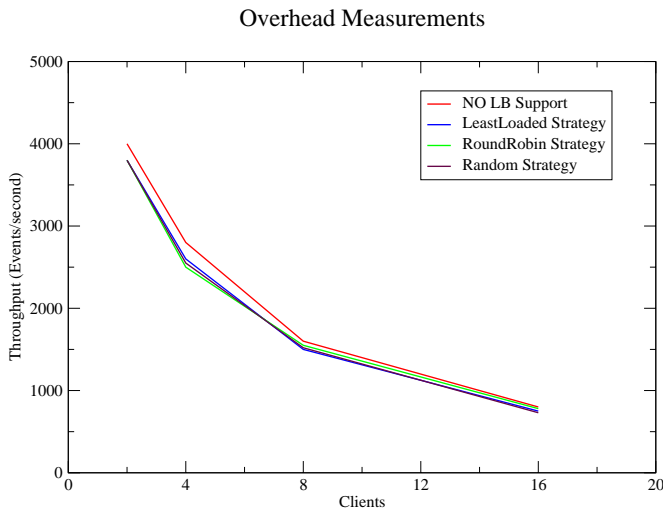


Figure 4: Overhead Measurements

percentage of the load experienced at the server at the particular moment, when the load balancing/shedding decision is made. The dampening factor can be chosen to range between 0 and 1. The higher the value, the lower the percentage of the newly reported load is taken into account.

Cygnus was configured to query the load monitor every 3 seconds instead of the 5 second default.

3.5 Overhead Measurements

Figure 1 shows how the client requests are initially handled by the load balancer component, before being bound to the appropriate servers. This section presents the results of the experiments that illustrate how this initial call to the load balancer incurs minimal overhead on the distributed application and hence does not affect the performance of the distributed application.

In these sets of experiments, the number of clients were varied between 2, 4, 8, and 16 and all clients were made 500,000 invocations on a single server. The experiments were repeated for the following set of configurations: (1) TAO Latency test with no load balancing support added, (2) TAO Latency test with load balancing support added and using the Round Robin load balancing strategy, (3) TAO Latency test with load balancing support added and using the Random load balancing strategy and (4) TAO Latency test with load balancing support added and using the Least Loaded load balancing strategy.

Figure 4 shows how client request throughput varies as the number of clients and servers are changed when using each of the configurations described above. The throughput experienced by the clients, using any of the strategies described above, decreased as the number of clients increased. This is

quite expected for any distributed application. The fact that the distributed application behaves the same even after adding the load balancing support built in the Cygnus load balancing and monitoring service illustrates that the load balancing service does not add any overhead that affects the performance of the distributed application.

The throughput experienced by the clients while using any of the strategies built in the Cygnus load balancing service is slightly less than the throughput experienced without adding the load balancing support. This slight decrease in the throughput is attributed to the initial call to the load balancer while binding the client to the appropriate server. The load balancer is thereafter not involved in any of the other invocations made by the client on the server, unless the server gets overloaded and the load balancer has to rebind the client to any other less loaded server. This case does not occur in these experiments, because of the properly chosen threshold values for the Least Loaded strategy runtime configuration.

The fact that the throughput values experienced are the same does not mean that we do not need to use the load balancing service at all. The primary use of the load balancer is to help in achieving improved scalability for a wide range of distributed applications and doing that without adding any extra overhead to the application infrastructure. Having proved that the load balancing service does not incur any extra overhead, we proceed to the next section where we illustrate how the different load balancing strategies built in the Cygnus load balancing service assist in improving the scalability for a wide range of distributed applications.

3.6 Scalability Results

The primary use of a load balancer is to improve scalability. As such, it is important to demonstrate that a particular load balancer configuration actually improves distributed application scalability. Three sets of benchmarks are shown below, one each for the Round Robin, Random, and Least Loaded load balancing strategies. Each set of benchmarks shows how throughput and latency vary as the number of clients is increased between 1 and 32 clients, and the number of servers is increased between 2 and 8 servers. All the experiments in these benchmarks are designed with all the clients generating uniform loads on the server. In general, only two or three server data sets are shown to illustrate trends without cluttering the benchmark graphs.

3.6.1 Round Robin Strategy Benchmarks

Figure 5 shows how client request throughput varies as the number of clients and servers are changed when using the Round Robin load balancing strategy. This figure shows how

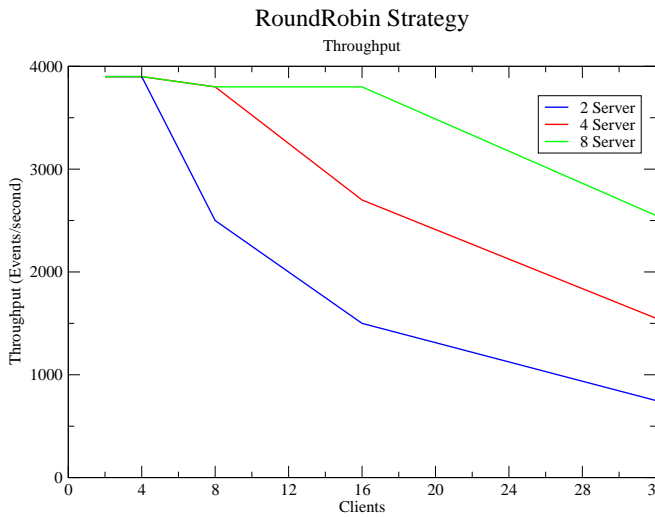


Figure 5: Round Robin Strategy Throughput

the throughput varies as the number of clients and the number of servers are changed. For example, throughput remained essentially unchanged as long as the number of clients was less than or equal to the number of servers. As the number of clients became larger than the number of servers, the throughput experienced by the clients became lesser and lesser. For example, when the number of servers is 4, the throughput experienced by 8 clients is less than the throughput experienced by 4 clients.

Figure 5 shows that the throughput experienced by the clients increased as the number of servers increased. For example, when the number of clients is 8, the throughput experienced when the number of servers is 4 is more than the throughput experienced when the number of servers is 2. These results show that the Cygnus load balancing/monitoring service helps the distributed application to scale well as the number of servers increase, *i.e.*, the increase in the number of servers does not incur extra overhead and the throughput increases as expected. Thus the Cygnus load balancing/monitoring service allowed access to increased number of servers without adding any extra overhead to the distributed application. Also, these results show that the Cygnus load balancing/monitoring service is able to handle increasing number of client requests by making use of the available servers in an efficient and transparent manner.

Figure 6 illustrates how request latency varied as the number of clients and servers were changed. This figure shows how employing Cygnus in the Latency performance test improved the latency. The latency experienced by the clients increased as the number of clients became larger than the number of servers. For example, when the number of servers is 4, the latency experienced by 4 clients is smaller than the latency

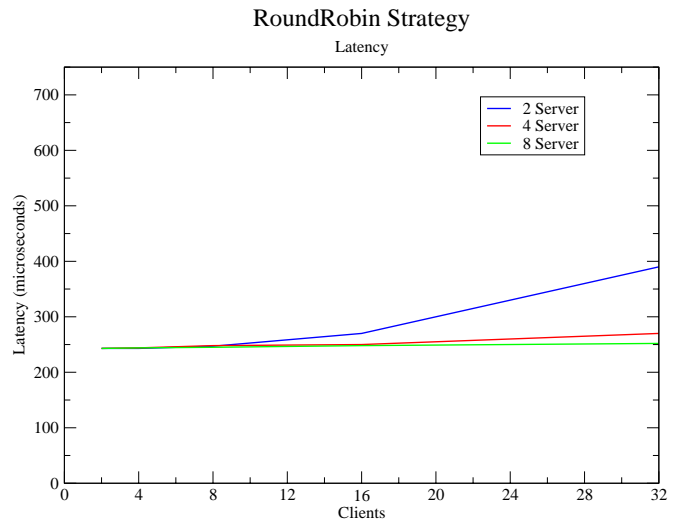


Figure 6: Round Robin Strategy Latency

experienced by 16 clients.

Increasing the number of servers improved the latency. For example, the latency for the 32 client and 2 server case is approximately 400 microseconds. Increasing the number of servers to 8 reduced the roundtrip latency to about 250 microseconds. This decrease in latency in turn enabled the servers to handle more requests and hence the throughput increased as more servers were added. The above claim is amply demonstrated in the Figure 5.

3.6.2 Random Strategy Benchmarks

Figure 7 depicts how the Random load balancing strategy implemented in Cygnus behaved when varying the number of clients and servers. This figure shows how the Random load balancing strategy behaves basically the same as the Round Robin load balancing strategy presented in Section 3.6.1. Both strategies exhibit similar scalability characteristics due to the fact that they are non-adaptive and use fairly simple member selection algorithms.

The results in Figure 7 do not mean, however, that all non-adaptive strategies will have the same throughput characteristics. It simply happens that in this case, client requests were distributed fairly equitably among the object group members chosen at random. Other cases could potentially result in multiple clients being bound to the same randomly chosen object group member. In those cases, and assuming that loads generated by all clients are uniform (as is the case in this test), throughput would be less than the Round Robin case because the strategy would not make use of all the available hardware resources. This could result in huge loss in money as we would waste the initial overcommitting of hardware resources.

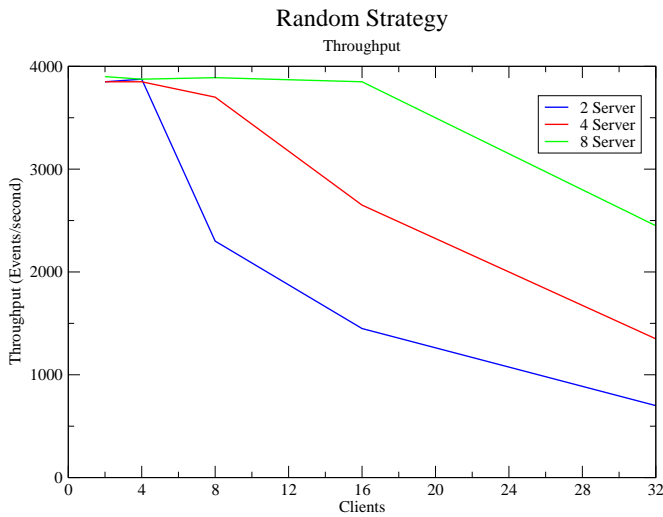


Figure 7: Random Strategy Throughput

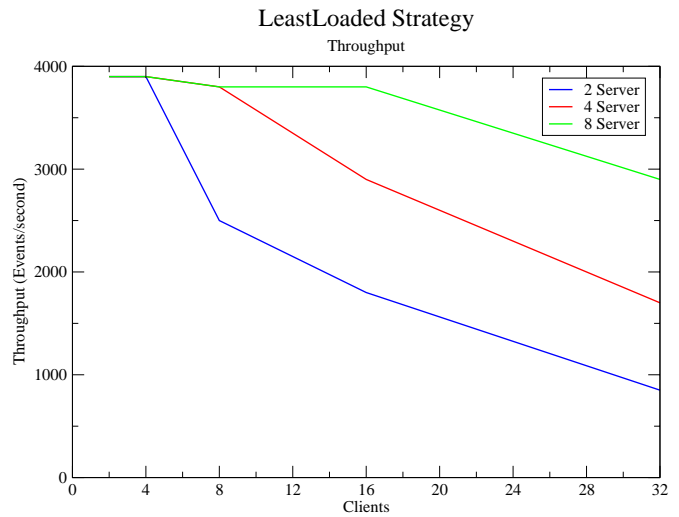


Figure 9: Least Loaded Strategy Throughput

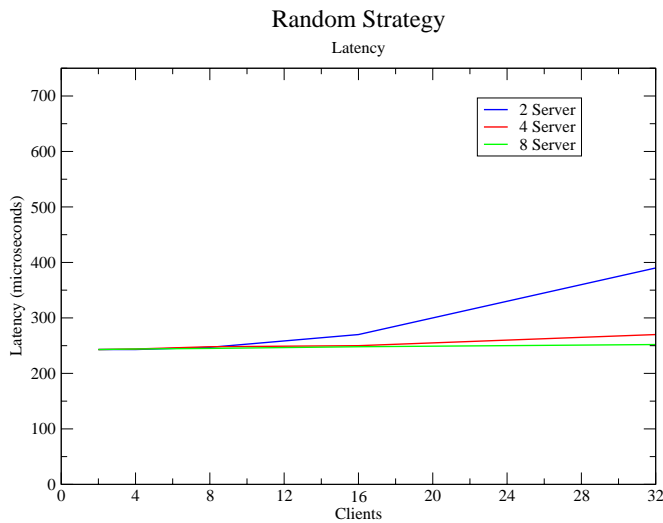


Figure 8: Random Strategy Latency

Figure 8 shows how the roundtrip latency characteristics for the Random load balancing strategy behaves the same way as the Round Robin load balancing strategy.

3.6.3 Least Loaded Strategy Benchmarks

The Least Loaded load balancing strategy used for this test configuration was designed to exercise the adaptive load balancing support in Cygnus explicitly. The runtime configuration described in section 3.4 was used in these benchmarking experiments.

Figure 9 illustrates how Cygnus' Least Loaded load balancing strategy reacts as the number of clients and servers are var-

ied. This figure illustrates how Cygnus' Least Loaded strategy results in a better performance for the distributed application, when compared to the Round Robin and Random load balancing strategies. For example, when the number of servers is 2 and the number of clients is 32, the throughput experienced by the clients while using the Least Loaded load balancing strategy is much more than the throughput experienced by the clients when using the Round Robin or Random load balancing strategies.

Figure 9 also illustrates that the overall system scalability improved. In particular, increasing the number of servers showed further improvements in scalability. For example, the throughput experienced by 16 clients when the number of servers is 2 is 1800 events/second, while the throughput obtained by 16 clients when the number of servers is 4 is 2800 events/second. So the Least Loaded strategy is able to make available increased number of servers to existing clients in a way that improves scalability without incurring any extra overhead.

The results shown in figure 9 is obtained mainly because of using the runtime configuration for Least Loaded strategy as discussed in section 3.4. The performance of the Least Loaded load balancing strategy mainly depends upon the values for the threshold variables being used by the experiments. This choice of values for the threshold variables varies between experiments and the value has to be determined by repeated running of the experiments. An improperly chosen value for the threshold variables may end up giving a performance as illustrated in figure 10.

As the results in figure 10 indicate, the performance of the Least Loaded strategy can be substantially less than the performance of the common non-adaptive load balancing strate-

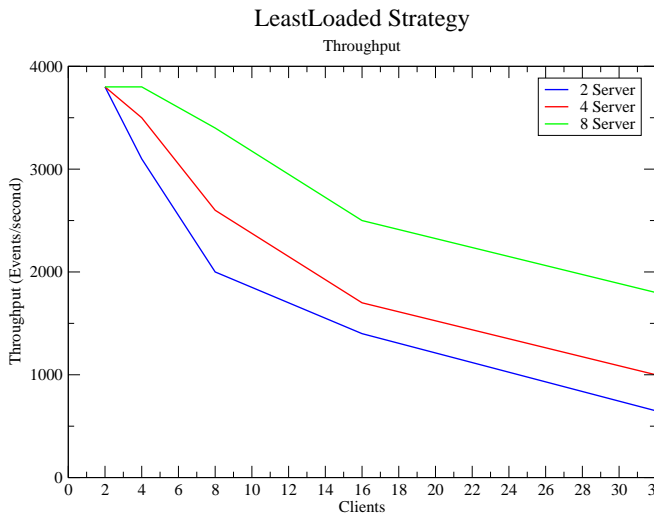


Figure 10: Least Loaded Strategy Throughput

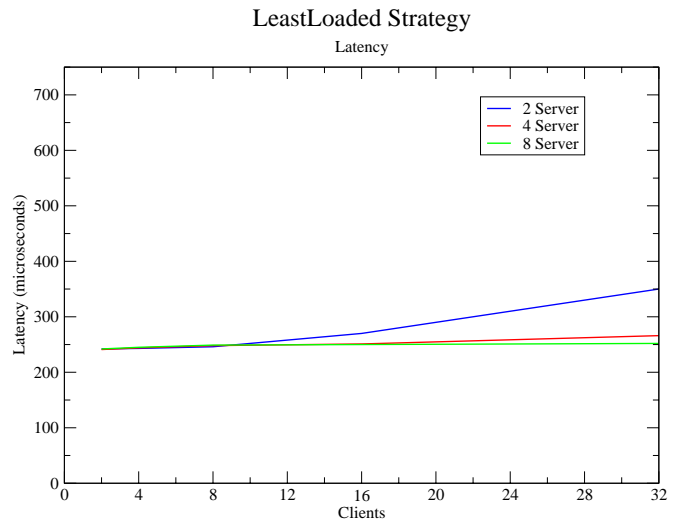


Figure 11: Least Loaded Strategy Latency

gies because of the improper values chosen for the threshold variables. For example, the throughput experienced by 32 clients when the number of servers is 2, when using the Least Loaded strategy is much less than the throughput experienced by the clients when using the common non-adaptive strategy like Round Robin strategy. This is because of the extra overhead incurred by the Least Loaded load balancing strategy over the common non-adaptive load balancing strategy like Round Robin strategy.

Cygnus's overhead is more apparent in the Least Loaded case when the number of clients is much larger than the number of servers. This overhead includes:

- Additional periodic requests on the server emanating from Cygnus when querying the server for its current load,
- Delays in client request binding as Cygnus waits for member loads to fall under a suitable value, *i.e.*, the reject threshold, and
- Request redirection incurred when servers forward requests back to Cygnus when their current load is over the configured critical threshold.

Despite the additional overhead, Figure 10 illustrates that overall system scalability still improved. More importantly, the throughput experienced by the clients did not change much when the number of clients is very less. For example, when comparing both the experiments when Least Loaded strategy was used, we can figure out that the throughput experienced by 4 clients when the number of servers is 4 is the same. The overhead gets more and more concrete, when the number of clients become larger and larger.

The latency results shown in Figure 11 illustrate reductions in roundtrip latency as the number of servers are increased. This figure shows how employing Cygnus in the Latency performance test improved the latency. The latency experienced by the clients increased as the number of clients became larger than the number of servers. For example, when the number of servers is 8, the latency experienced by 4 clients is smaller than the latency experienced by 16 clients.

Increasing the number of servers improved the latency. For example, the latency for the 32 client and 2 server case is approximately 350 microseconds. Increasing the number of servers to 8 reduced the roundtrip latency to about 250 microseconds. This decrease in latency in turn enabled the servers to handle more requests and hence the throughput increased as more servers were added. The above claim is amply demonstrated in the Figure 9.

3.7 Performance Comparison

Section 3.6 and section 3.5 showed that the non-adaptive and adaptive CORBA LB/M strategies (*i.e.*, *Round Robin*, *Random*, and *Least Loaded*) supported by Cygnus can be quite effective in increasing overall scalability of CORBA-based distributed applications. The strategy configurations used in these benchmarks caused the Least Loaded adaptive load balancing strategy benchmark to have similar throughput and latency as their non-adaptive counterparts when the clients generate uniform loads. These results also demonstrate that the load monitor component added as part of the Least Loaded strategy incurs low overhead when reporting the load information to the load balancer. The overhead experienced while using the load balancer is due to the initial call to the load balancer and the

immediate forwarding of the calls from the clients to their respective servers.

Given a test configuration with clients generating non-uniform loads, the benefits of adaptive load balancing would be more evident. The benchmarking experiments were modified to make the clients generate non-uniform loads. The following experiments are designed to compare the performance of the adaptive and non-adaptive load balancing strategies built in Cygnus load balancing and monitoring service. Round Robin strategy is used as an example for non-adaptive load balancing strategy while Least Loaded strategy is used as an example for adaptive load balancing strategy.

The benchmarking experiments were again based on the single-threaded form of the “Latency” performance test distributed with the TAO open-source software release. The benchmarking experiments were modified with the following variations in the Latency test:

1. **Latency test with Round Robin load balancing strategy with clients generating non-uniform loads.** In this benchmark, the Latency test was configured to employ the Round Robin load balancing strategy to improve scalability. Half of the clients were made to just invoke the intended operation on the server, while half of the clients were made to burn some CPU time before making the actual invocation on the server. This ensured that the load generated by the clients on the servers is not uniform. Half of the clients were made to invoke 100,000 iterations while half of the clients were made to invoke 500,000 iterations.
2. **Latency test with Least Loaded load balancing strategy with clients generating non-uniform loads.** This final benchmark configuration uses Cygnus’ Least Loaded load balancing strategy to improve scalability. A request monitor was added that queried the servers for the total number of requests being handled per second by the server at that particular moment. The request monitor queried the server every 3 seconds. Half of the clients were made to just invoke the intended operation on the server, while half of the clients were made to burn some CPU time before making the actual invocation on the server. Half of the clients were made to invoke 100,000 iterations while half of the clients were made to invoke 500,000 iterations. This ensured that the load generated by the clients on the servers is not uniform.

3.7.1 Number of clients equal or very much larger than number of servers

The following experiments analyze the performance of the adaptive and non-adaptive load balancing strategies when the

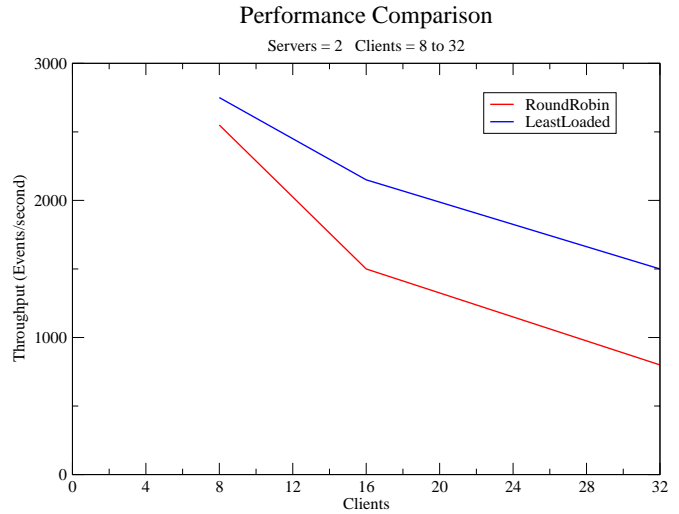


Figure 12: Throughput Comparison

number of servers is varied between 2 and 8 and the number of clients is varied between 8 and 32.

Figure 12 shows how client request throughput varies as the number of clients ranges between 8 and 32 and the number of servers is 2. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the experiments. As the results indicate, the performance of the adaptive load balancing strategy is much better than the performance of the non-adaptive load balancing strategy. As the difference between the number of servers and the number of clients increases, the throughput difference between the throughput achieved by adaptive load balancing strategy and the throughput achieved by non-adaptive load balancing strategy also increases. For example, the throughput difference between the adaptive load balancing strategy and the non-adaptive load balancing strategy, when the number of clients is 8 is about 200 while the throughput difference when the number of clients is 32 is about 800.

Figure 13 shows how client request throughput varies as the number of clients ranges between 8 and 32 and the number of servers is 4. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the experiments. As the results indicate the performance of both the strategies when the number of servers is 4 is very much similar to the performance of the strategies when the number of servers is 2. More importantly, irrespective of the number of clients, the throughput difference between the strategies is much lesser in the 4-server case than in the 2-server case.

Figure 14 shows how client request throughput varies as the number of clients ranges between 8 and 32 and the number of servers is 8. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the exper-

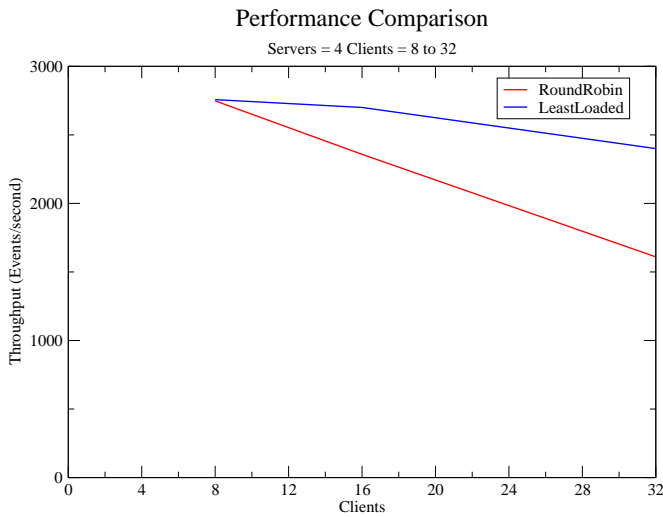


Figure 13: Throughput Comparison

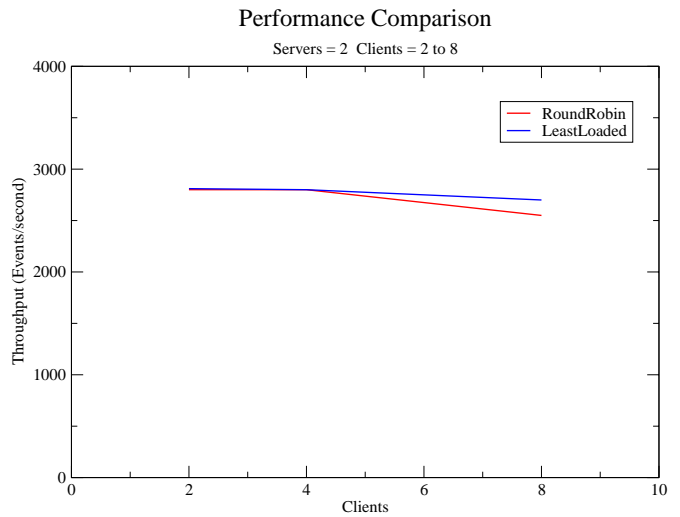


Figure 15: Throughput Comparison

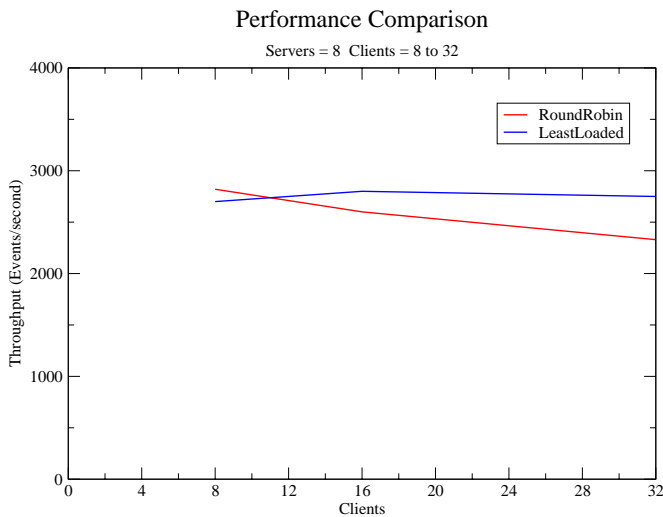


Figure 14: Throughput Comparison

iments. As the results indicate the performance of both the strategies when the number of servers is 8 is very much similar to the performance of the strategies when the number of servers is 2. More importantly, irrespective of the number of clients, the throughput difference between the strategies is much lesser in the 4-server case than in the 2-server case. Also the point to be noted, is that when the number of clients is 8, the throughput obtained by adaptive load balancing strategy is slightly lesser than the throughput obtained through non-adaptive load balancing strategy.

The results from the three figures indicate that the performance of the adaptive load balancing strategy is much better than the performance of non-adaptive load balancing strategy,

when the number of clients is very much larger than the number of servers. As the difference between the number of clients and the number of servers increase, the performance of the adaptive load balancing strategy keeps increasing.

3.7.2 Number of clients less than, equal or slightly more than the number of servers

The following experiments analyze the performance of the adaptive and non-adaptive load balancing strategies when the number of servers is varied between 2 and 8 and the number of clients is varied between 2 and 8.

Figure 15 shows how client request throughput varies as the number of clients ranges between 2 and 8 and the number of servers is 2. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the experiments. The results indicate that the performance of both the strategies are exactly similar when the number of clients is ranging between 2 and 6. There is a slight performance difference when the number of clients is 8. This shows that there is nothing to choose between adaptive and non-adaptive load balancing strategies when the number of clients is very much equal or slightly greater than the number of servers.

Figure 16 shows how client request throughput varies as the number of clients ranges between 2 and 8 and the number of servers is 4. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the experiments. The results indicate that the performance of both the strategies are exactly similar irrespective of the number of clients.

Figure 17 shows how client request throughput varies as the number of clients ranges between 2 and 8 and the num-

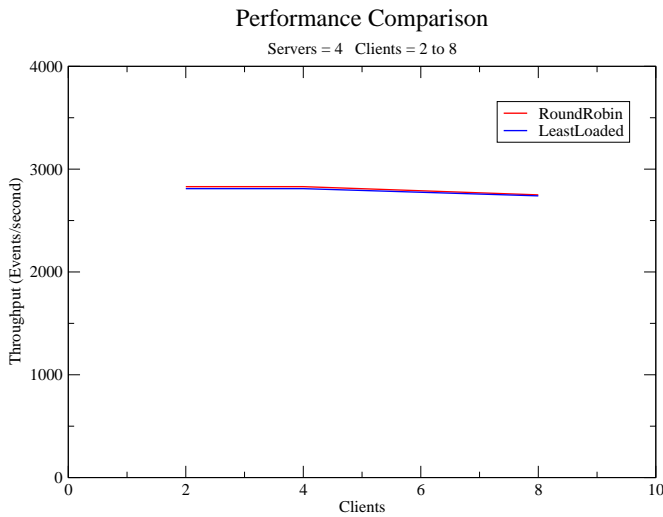


Figure 16: Throughput Comparison

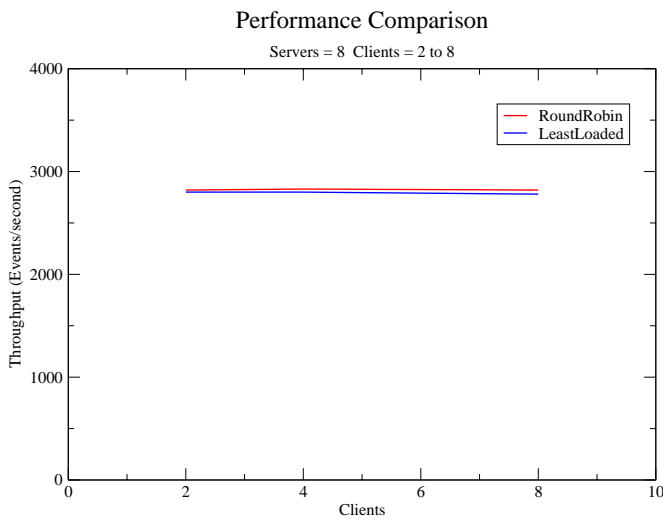


Figure 17: Throughput Comparison

ber of servers is 8. The results are shown for both the adaptive and non-adaptive load balancing strategies used in the experiments. The results indicate that the performance of both the strategies are exactly similar irrespective of the number of clients.

3.8 Lessons Learned

3.9 Summary of Empirical Results

The goal of the experiments in this paper was to show the extent to which employing a CORBA-compliant LB/M implementation, such as Cygnus, can improve distributed applica-

tion scalability without adding any extra overhead to the infrastructure. As our results show, scalability was indeed improved in all test cases.

4 Related Work

This section describes other efforts on load balancing at various levels of abstraction and compares and contrasts our research on middleware-based load balancing and Cygnus with representative related work.

4.1 Load Balancing at Various Levels of Abstraction

A significant amount of work has been done on load balancing services at the network, the operating system, and middleware levels, as described below.

4.1.1 Network-based Load Balancing

Network-based load balancing services make decisions based on the frequency at which a given site receives requests [12]. For example, routers [1] and DNS servers often perform network-based load balancing, as depicted in Figure 18. Load

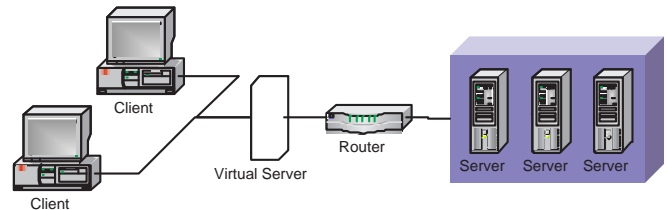


Figure 18: Network-level Load Balancing

balancing performed at the network level has the disadvantage that load balancing decisions are based solely on request destinations rather than request content.

4.1.2 OS-based Load Balancing

Load Balancing has been supported at the operating system level via process migration and memory ushering [13]. Using process migration, overloaded processes can be shifted to less loaded hosts to reduce overall execution time, as shown in Figure 19. These hosts can be present locally or in a remote workstation. Depending on the location, migrated processes can be scheduled immediately or after a slight delay. Process migration decisions can be made using various metrics, including node workload [14, 15], job memory utilization [16], and CPU-usage [17]. Load balancing at the operating system

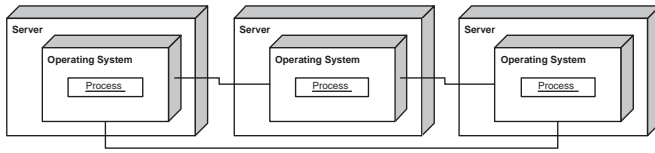


Figure 19: OS-level Load Balancing

level [18, 19, 2] has the advantage of performing the balancing at multiple levels (*i.e.*, at both the network and operating system levels) that are largely transparent to applications.

OS-level load balancing suffers from many of the same problems that network-based load balancing does, such as inflexible load metric selection and not being able to take advantage of request content. OS-based load balancing may also be too coarse-grained for some applications where it is the server process itself, rather than the object residing within the server, that must be load balanced. In addition, load balancing via process migration has several disadvantages. For instance, when processes are migrated to a new node, the resources they used at the previous node may not be available, which may cause significant reduction in the execution time. MOSIX [13] solves this problem by contacting the user’s “home” node for interactions and resource use. However, frequent remote calls to the user’s home can increase network traffic significantly, thereby obviating the benefits of load balancing.

4.1.3 Middleware-based Load Balancing

Middleware-based load balancing provides significant flexibility in terms of influencing how a load balancing service makes decisions, and in terms of applicability to different types of applications [20, 21]. In particular, only middleware-based load balancing is effective for *distributed* applications, for example, since it is able to distributed system behavior and state into account. As shown in Figure 20, middleware-based load balancing enables flexible application-defined selection of load metrics, in addition to the ability to make load balancing decisions based on request content.

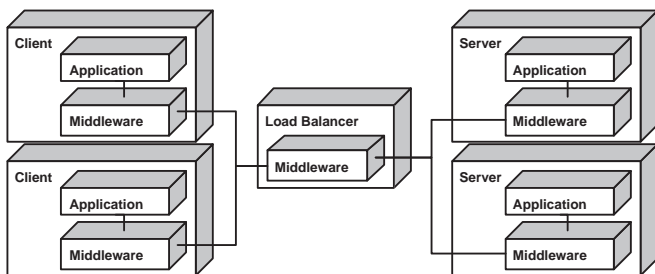


Figure 20: Middleware-level Load Balancing

Some middleware implementations [22, 23] integrate load balancing functionality into the object request broker (ORB) level of the middleware itself, whereas others [24, 25, 26, 27, 28] implement load balancing support at a higher level, such as the common middleware services level. The latter is the approach taken by the Cygnus load balancing and monitoring service described in Section 2. The remainder of this section compares and contrasts our work on middleware load balancing and Cygnus with representative related work.

4.1.4 CORBA Load Balancing

A number of projects focus on CORBA load balancing, which can be implemented at the following levels in the OMG middleware reference architecture.

Service-level Load balancing can 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 via 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 load balancing is performed to distribute consumers evenly. In contrast, TAO’s load balancing service is more general and can be used both for event services and for requests on application-defined CORBA objects.

[25] extends the CORBA Naming Service to add load balancing capabilities. When a client calls the `NamingContext::resolve()` operation it receives the IOR of the least-loaded server registered with the given name. This approach, however, introduces a problem in which many servers may register themselves with the same name and this practice is not standards-compliant. In contrast, Cygnus uses its `LoadBalancer` component to provide clients with the IOR of the next available least-loaded server, which ensures there are no naming conflicts with the available servers.

[26] presents a load balancing service for CORBA-based applications that is analogous to the CORBA Trading Service. Their load balancing service uses an independent central component that monitors and balances the loads. Since their load balancer is a centralized component it is not only a single-point of failure but is also not fully scalable, *i.e.*, the service can become overloaded and result in significant delays. In contrast, the Cygnus load balancing service has a load monitor component that monitors loads on a given resource. This component implements the Strategy and Mediator patterns [8] to define an interface that can be used to (1) report loads to the load balancer or (2) obtain loads from the load monitor. The load monitor component minimizes the amount of communication between the servers and the load balancer so that there is a loose coupling between them, thereby providing a load balancer that is independent of the selected load metric and to

minimize the amount of work performed by the load balancer component.

Various commercial CORBA implementations provide service-level load balancing. For example, IONA's Orbix [27] can perform load balancing using the CORBA Naming Service. Different group members 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 [4]. BEA's WebLogic [28] uses a per-request load balancing strategy, also described in [4]. In contrast, TAO's load balancing service Cygnus does not incur the per-request network overhead of the BEA strategy, yet can still adapt to dynamic changes in the load, unlike static load balancing services, such as Orbix's Naming Service implementation.

ORB-level. Load balancing can also 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 instance, Inprise's VisiBroker implements an ORB-level load balancing strategy, where Visibroker's object adapter [23] creates object references that point to Visibroker's Implementation Repository (called the OSAgent) that plays the role of an activation daemon and a load balancer.

The advantage of ORB-level techniques is that the amount of indirection involved when balancing loads can be reduced because load balancing mechanisms are closely coupled with the ORB *i.e.*, the length of communication paths is shortened. The disadvantage of ORB-level load balancing, however, is that it requires modifications to the ORB itself, so until such modifications are adopted by the OMG, they will be proprietary, which reduces their portability and interoperability. The Cygnus service-level load balancer therefore does not rely on ORB-level extensions or non-standard features, *i.e.*, it does not require any modifications to TAO's ORB core or object adapter. Instead, it takes advantage of standard mechanisms in CORBA 3.0 to implement adaptive load balancing. Unlike ORB-based load balancing approaches, however, Cygnus uses only standard CORBA features, so it can be ported to any C++ CORBA ORB that implements the CORBA 3.0 or newer specification.

5 Concluding Remarks

Middleware-based load balancing is an important technology for improving the scalability of distributed applications. This paper analyzes the performance of Cygnus, which is a

middleware-based load balancing and monitoring (LB/M) service capable of performing both non-adaptive and adaptive load balancing. Our results show how Cygnus allows distributed applications to be load balanced adaptively and efficiently. Cygnus increases the scalability of distributed applications by distributing requests across multiple back-end server members without increasing round-trip latency substantially or assuming predictable, or homogeneous loads. The empirical results in Section 3 show that introducing the Cygnus LB/M service into distributed applications can substantially improve scalability while incurring minimal run-time overhead. As a result, developers can concentrate on their core application behavior, rather than wrestling with complex middleware mechanisms needed to make their distributed applications scalable.

While our work on Cygnus has shown that it is a strong scalability solution for middleware-based distributed applications, there are a number of enhancements that can be made. Our ongoing work on Cygnus therefore involves the following topics:

- *Support for stateful object group members.* Load balancing of *stateless* objects may not always be possible or the best choice, but load balancing of *stateful* objects is non-trivial.
- *Decentralized load balancing.* The current *centralized* design introduces a single point of failure that may affect reliability and scalability of the load balancer itself.
- *Fault tolerant load balancing.* Load balancers are often used in high availability systems with stringent fault tolerance requirements. Determining how to improve load balancer fault tolerance characteristics is essential prior to wide scale deployment.
- *Self-adaptive Load Balancing Strategies.* Adaptive load balancing strategies may not perform as well if not configured properly, but knowing which configuration is best is difficult. Self-adaptive load balancing strategies that alter their configuration on-the-fly to better handle non-deterministic load conditions are key for ease of deployment and maximizing scalability.
- *Object group conflicts.* Multiple object group members from different object groups residing at the same location may have potential resource conflicts due to incompatible load balancing strategies configured for each group. It is conceivable that such a scenario will occur in a real world application, meaning that learning how to deal with this problem is important.

These issues will be addressed as our work on Cygnus progresses.

References

- [1] Cisco Systems, Inc., “High availability web services,” www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/mnibm_wp.htm, 2000.
- [2] Werner G. Krebs, “Queue Load Balancing / Distributed Batch Processing and Local RSH Replacement System,” www.gnuqueue.org/home.html, 1998.
- [3] Douglas C. Schmidt, Rick Schantz, Mike Masters, Joseph Cross, David Sharp, and Lou DiPalma, “Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems,” *CrossTalk*, Nov. 2001.
- [4] Ossama Othman, Carlos O’Ryan, and Douglas C. Schmidt, “Strategies for CORBA Middleware-Based Load Balancing,” *IEEE Distributed Systems Online*, vol. 2, no. 3, Mar. 2001.
- [5] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, Dec. 2002.
- [6] Ossama Othman, Jaiganesh Balasubramanian, and Douglas C. Schmidt, “The Design of an Adaptive Middleware Load Balancing and Monitoring Service,” in *LNCS/LNAI: Proceedings of the Third International Workshop on Self-Adaptive Software*, Heidelberg, June 2003, Springer-Verlag.
- [7] Ossama Othman and Douglas C. Schmidt, “Optimizing Distributed system Performance via Adaptive Middleware Load Balancing,” in *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, Snowbird, Utah, June 2001, ACM SIGPLAN.
- [8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
- [9] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*, Wiley & Sons, New York, 2000.
- [10] Ossama Othman, Carlos O’Ryan, and Douglas C. Schmidt, “Designing an Adaptive CORBA Load Balancing Service Using TAO,” *IEEE Distributed Systems Online*, vol. 2, no. 4, Apr. 2001.
- [11] Khanna, S., et al., “Realtime Scheduling in SunOS 5.0,” in *Proceedings of the USENIX Winter Conference*. 1992, pp. 375–390, USENIX Association.
- [12] Ervin Johnson and ArrowPoint Communications, “A Comparative Analysis of Web Switching Architectures,” www.arrowpoint.com/solutions/white_papers/ws_archv6.html, 1998.
- [13] La’adan. O Barak. A and Shiloh. A., “Scalable Cluster Computing With MOSIX for Linux,” *Journal of Future Generation Computer Systems*, pp. 361–372, May 1998.
- [14] Ron Lavi and Amnon Barak, “The Home Model and Competitive Algorithms for Load Balancing in a Computing Cluster,” in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS’ 01)*, Phoenix, USA, Apr. 2001, IEEE.
- [15] Chi-Chung Hui and Samuel T. Chanson, “Improved Strategies for Dynamic Load Balancing,” *IEEE Concurrency*, vol. 7, no. 3, July 1999.
- [16] Leonardi. L Corradi. A and Zambonelli. F, “Diffused load-balancing policies for Dynamic applications,” *IEEE Concurrency*, vol. 7, no. 1, July 1999.
- [17] M. Harchol-Balter and A. Downey., “Exploiting process lifetime distributions for dynamic load balancing,” *ACM Transactions on Computer Systems*, vol. 15, pp. 253–285, Mar. 1997.
- [18] 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.
- [19] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling, “Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs,” in *Proceedings, IEEE Aerospace*. IEEE, 1997.
- [20] T. Ewald, “Use Application Center or COM and MTS for Load Balancing Your Component Servers,” www.microsoft.com/msj/0100/loadbal/loadbal.asp, 2000.
- [21] Vittorio Ghini, Fabio Panzieri, and Marco Roccetti, “Client-centered Load Distribution: A Mechanism for Constructing Responsive Web Services,” in *Proceedings of the 34th Hawaii International Conference on System Sciences - 2001*, Hawaii, USA, 2001.
- [22] Markus Lindermeier, “Load Management for Distributed Object-Oriented Environments,” in *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, Sept. 2000, OMG.

- [23] Inc. Inprise Corporation, “VisiBroker for Java 4.0: Programmer’s Guide: Using the POA,” www.inprise.com/techpubs/books/vbj/vbj40/programmers-guide/poa.html, 1999.
- [24] Key Shiu Ho and Hong Va 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’00)*, Antwerp, Belgium, Sept. 2000, OMG.
- [25] Thomas Barth, Gerd Flender, Bernd Freisleben, and Frank Thilo, “Load Distribution in a CORBA Environment,” in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA’99)*, Edinburgh, Scotland, Sept. 1999, OMG.
- [26] Markus Aleksy, Axel Korthaus, and Martin Schader, “Design and Implementation of a Flexible Load Balancing Service for CORBA-based Applications,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’01)*, Las Vegas, USA, June 2001, IEEE.
- [27] IONA Technologies, “Orbix 2000,” http://www.iona.com/products/orbix2000_home.htm.
- [28] BEA Systems Inc., “WebLogic Administration Guide,” edoc.bea.com/wle/.