The Design and Performance of an Adaptive Middleware Load Balancing and Monitoring Service

Ossama Othman and Jaiganesh Balasubramanian
{ossama,jai}@doc.ece.uci.edu
Dept. of Electrical
and Computer Engineering
University of California
608 Engineering Tower
Irvine, CA 92697, USA

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
Institute for Software
Integrated Systems
Vanderbilt University
2015 Terrace Place
Nashville, TN 37203, USA

Abstract

Middleware is increasingly used as the infrastructure for applications with stringent quality of service (QoS) requirements, including scalability. One way to improve the scalability of distributed applications is to use adaptive middleware to balance system processing load dynamically among multiple servers. Adaptive middleware load balancing can help improve overall system performance by ensuring that client application requests are distributed and processed equitably across groups of servers.

This paper presents the following contributions to research on adaptive middleware load balancing techniques: (1) it describes deficiencies with common load-balancing techniques, such as introducing unnecessary overhead or not adapting dynamically to changing load conditions, (2) it describes the capabilities of Cygnus, which is an adaptive load balancing service, and (3) it presents the results of empirical benchmarks that systematically evaluate different load balancing strategies provided in Cygnus by measuring their scalability showing how well each strategy balances system load. The findings in this paper show that adaptive middleware load balancing is a viable solution for improving the scalability of distributed applications.

1 Introduction

Motivation. As the demands of resource-intensive distributed applications have grown, the need for improved overall scalability has also grown. For example, client requests may arrive dynamically–not deterministically–in many distributed applications, such as automated stock trading, e-commerce transactions, and total ship computing environments. Moreover, the amount of load incurred by each request may not be known in advance.

These conditions require that a distributed application be able to redistribute requests dynamically. Otherwise, one or more backend servers may potentially become overloaded, whereas others will be underutilized. In other words, the system must *adapt* to changing load conditions. In theory, applying adaptability in conjunction with multiple backend servers can

- Allow the system to scale up gracefully to handle more clients and processing workload in larger configurations.
- Reduce the initial investment when the number of clients is small and
- Increase the reliability of the overall system, *e.g.*, by redirecting requests to replicated servers when failures occur.

Achieving this degree of scalability requires a sophisticated load balancing service. Ideally, this service should be transparent to existing distributed application 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.

Evaluating candidate solutions. Load balancing can be performed at the network, operating system, middleware, or application layers, as shown in Figure 1. Network-level load balancing is often provided by routers and domain name servers [5]. OS-level load balancing is generally provided by clustering software [19]. Application-level load balancing is performed by the application itself [1]. A layer may take advantage of load balancing in layers below it when balancing loads at its level. For instance, application-level load balancing may employ load balancing facilities supplied by the OS.

While load balancing can be performed in the layers outlined above, these layers have the following disadvantages that can make them unsuitable for use in distributed applications that require dynamic adjustment to runtime load conditions:

1. The inability to take into account client request content
2. Lack of transparency and
3. High maintenance lifecycle costs.

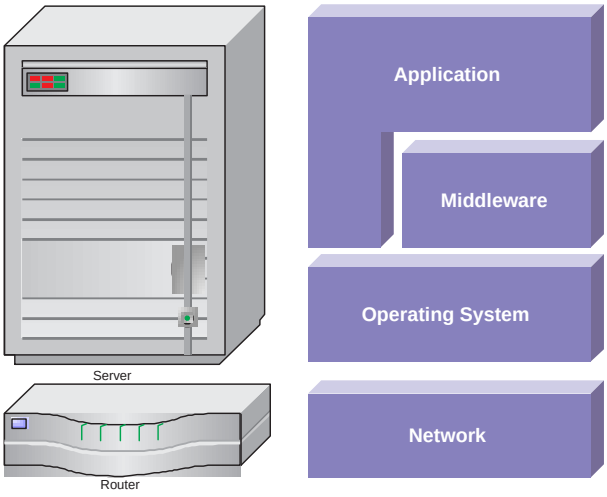


Figure 1: Load Balancing Layers

In particular, network- and OS-based load balancing suffer from the first disadvantage, *i.e.*, they cannot take into account client request content because that information is necessarily application-specific. Application-based load balancing suffers from the last two disadvantages, *i.e.*, transparency is lost since the application itself must be modified to support load balancing, which can complicate code development and maintenance.

Given these deficiencies, a cost-effective way to address the application demands listed above is to employ load balancing services based on distribution *middleware* [20], such as CORBA [16] or Java RMI [23]. These load balancing services distribute client workload equitably among various backend servers to obtain improved response times and scalability.

Earlier generations of middleware load balancing services largely supported simple, centralized distributed application configurations. For example, stateless distributed applications that require load balancing often integrate their load balancing service with a naming service [3, 11]. In this approach, a naming service returns a reference to a different object each time it is accessed by a client. Implementing a load balancing service via a naming service can be (1) *overly static*, *e.g.*, if the naming service does not consider dynamic load conditions when returning an object reference to its clients and/or (2) *inefficient*, *e.g.*, due to the extra (and ultimately unnecessary) levels of indirection and round-trip latencies.

In contrast, *adaptive* middleware load balancing services that consider dynamic load conditions when making decisions can yield the following benefits:

- An adaptive load balancing service can support a larger range of distributed systems since it need not be designed for a specific application, *i.e.*, it is more flexible.
- From the load balancing service implementation perspective, since a single load balancing service can be used for many types of applications, the effort needed to develop a load balancing service for a specific application is reduced. This generally allows for simpler and better load balancing service implementations.
- It is possible to concentrate on the load balancing service in general, rather than a particular aspect geared solely to one application, which can improve the quality of optimizations used in the load balancing service.

Unfortunately, first-generation adaptive middleware load balancing services [13, 10], including our own earlier work [18, 17] on the topic, do not provide solutions for key dimensions of the problem space. In particular, they provided insufficient functionality to satisfy advanced distributed application requirements, such as the ability to tolerate faults, install new load balancing algorithms at run-time, and create group members on-demand to handle bursty clients. The lack of support for this advanced functionality in first-generation adaptive middleware load balancers has impeded distributed system scalability. Moreover, the lack of *standardized* interfaces and policies have precluded reuse of interoperable off-the-shelf adaptive middleware load balancing services. This paper therefore explores a previously unexamined dimension in the middleware space: *the design and performance of a scalable adaptive load balancing service based on the OMG CORBA standard*.

Our work in this paper is presented in the context of one of the OMG *Load Balancing and Monitoring* (LB/M) service specification proposals [15] and our Cygnus implementation of this service that guided the proposal effort. Though CORBA has standardized solutions for many distributed system challenges, such as predictability, security, transactions, and fault tolerance, it does not yet have specify how to tackle load balancing capabilities required by distributed systems architects and developers. Cygnus is available with *The ACE ORB* (TAO) [21] version 5.3, which implements the CORBA 3.0 specification [16]. The software, documentation, examples, and benchmarking tests for TAO and Cygnus are open-source and can be downloaded from `deuce.doc.wustl.edu/Download.html`.

Paper organization. The remaining sections of this paper are organized as follows: Section 2 describes the proposed CORBA Load Balancing and Monitoring (LB/M) service specification and the architecture of Cygnus, which is our LB/M service implementation; Section 3 presents benchmarks that quantitatively evaluate how the Cygnus adaptive middleware LB/M architecture improves distributed application scalability; Section 4 describes other R&D efforts that are related to load balancing; and Section 5 presents concluding remarks.

2 Cygnus: An Adaptive Middleware Load Balancing and Monitoring Service

This section motivates and describes the key components and capabilities of Cygnus, which is the open-source middleware framework integrated with TAO that guided the design of our proposed OMG CORBA Load Balancing and Monitoring (LB/M) service specification [15]. Sidebar 1 defines and illustrates the load balancing concepts and components[1] used throughout this paper and the OMG LB/M proposal. TAO and Cygnus implement all the components shown in the figure in Sidebar 1. TAO facilitates location-transparent communication between (1) clients and instances of the Cygnus load balancer, (2) a load balancer and the object group members, and (3) clients and the object group members. Cygnus also keeps track of which members belong to each object group.

2.1 Overview of the Cygnus Load Balancing Model

In contrast to load balancing models that are process-oriented (where loads are balanced between processes) or object-oriented (where loads are balanced between objects), the load balancing model employed by Cygnus is *location-oriented*. For non-adaptive Cygnus load balancing strategies, the member to receive the next client request is based on the *location* where a specific member of an object group resides. The adaptive Cygnus load balancing case differs in that member selection is performed based on the loads at a given *location*. In both cases, neither process nor object characteristics are necessarily used when making load balancing decisions.

Although hosts are often associated with locations, the location-oriented model used in Cygnus makes no assumptions about the application's interpretation of what a "location" is. For example, an application could decide to associate a process with a location instead of a host. The load balancing model would still be location-oriented in this case, however, since the load balancer would not be aware that the location was actually a process.

The deployed structure of the location-oriented load balancing service is shown in Figure 2. The Cygnus load balancing model allows members from different object groups to reside at the same location. For instance, a member from `Group 1` and a member from `Group N` can each reside at a single location. This flexibility is one of the strengths of the Cygnus load balancing model when compared with earlier adaptive load balancing approaches [13, 10, 18, 17]. Additional flexibility can be found in Cygnus' support for object group-specific properties, such as the load balancing strategy in use.

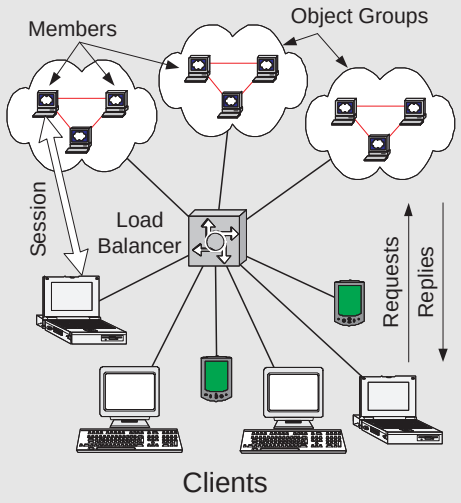
[1] In this paper, the term *component* is used generically, *i.e.*, an identifiable entity in a program, rather than more specifically, *e.g.*, a component in the CORBA Component Model [14].

Sidebar 1: Key Load Balancing Concepts

The key load balancing concepts and components used in this paper 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.

The following figure illustrates the relationships between these components:



2.2 Resolving Load Balancing Challenges with Cygnus

Figure 3 illustrates the relationships among the components in the Cygnus. As shown in this figure, the Cygnus adaptive LB/M middleware service consists of the (1) *load man-*

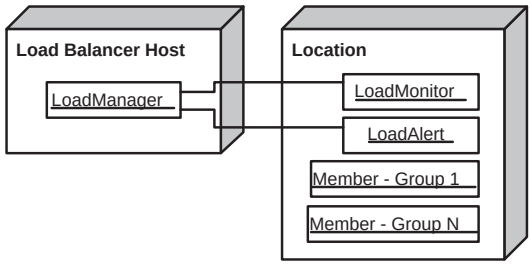


Figure 2: Deployed Structure of the Location-Oriented CORBA Load Balancing Service.

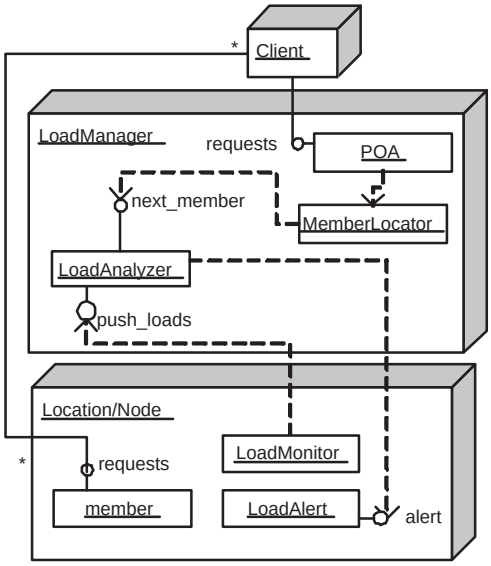


Figure 3: Components in the Cygnus LB/M Service

ager, which is the application entry point for all load balancing tasks, (2) *member locator*, which is the load balancing component responsible for binding a client to a member, (3) *load analyzer*, which analyses load conditions and triggers load shedding when necessary, (4) *load monitor*, which makes load reports available to the load manager, and (5) *load alert*, which is a component through which load shedding is performed.

Although the preceding discussion and Figure 3 outline the elements of the Cygnus, they do not motivate what these elements do or more importantly *why* they are important. The remainder of this section therefore justifies the need for these elements by explaining the key challenges they address, which include:

1. Extensible load analysis and shedding
2. Flexible load reporting and
3. Facilitating transparent and scalable load shedding.

For each challenge, we describe (1) how a particular component of Cygnus resolves problems that arise when balancing workloads in a middleware context and (2) how load balancing and monitoring is implemented in Cygnus. Our primary focus is on the use of adaptivity to enhance scalability.[2] As discussed below, the Cygnus load manager enables clients and servers to participate in load balancing decisions without unduly exposing them to tasks that can and should remain internal to the load balancing service. The member locator allows a load balancer to *transparently* inform a client that it should issue requests to a chosen object group member.

Other LB/M implementations, such as the one found in Orbix 2000 [11], employ concepts similar to the ones described below. Those implementations are less flexible than the approach employed by Cygnus, however, and do not separate concerns as cleanly.

Challenge 1: Extensible Load Analysis and Shedding

Context. The same load balancing service is used to balance loads for multiple (potentially different) distributed applications.

Problem. Load balancing multiple distributed applications with different resource requirements can be done in at least two ways:

- Create a different load balancing service instance for each type of distribute application. This solution, however, is hard to maintain. For example, when a new distributed application is deployed, a new load balancing service must be started and configured, which is logistically complex and costly.
- Use a single shared load balancing service instance to manage loads for multiple applications with different resource requirements. This solution requires that the load balancing service be extensible enough to allow run-time configuration of the load analysis and shedding mechanism on a per-object group basis, which is one of the requirements set forth in [17].

Solution → Load analyzer. Define a load analyzer component that decides which member will receive the next client request. The load analyzer also allows a load balancing strategy to be selected explicitly at run-time, while still maintaining a simple and flexible design. Since the load balancing strategy can be chosen at run-time, member selection can be tailored to fit the dynamics of a system that is being load balanced. An additional task the load analyzer performs is to initiate load shedding at locations where deemed necessary. This task only occurs when using an adaptive load balancing strategy.

Implementing the load analyzer in Cygnus. Cygnus implements the load analyzer component as a logical entity, *i.e.*, an actual load analyzer component does not exist, though

[2] Portability and transparency issues addressed by the load manager and member locator components are beyond the scope of this paper.

Cygnus functions as if one did exist. In particular, the tasks performed by the load analyzer are handled by objects that implement load balancing algorithms and are registered with Cygnus. Cygnus uses an implementation of the Strategy [7] design pattern to achieve this functionality. Load balancing strategies are registered with Cygnus as CORBA object references, meaning that load balancing strategy implementations may actually reside at remote locations.

Load balancing strategies can invoke adaptive load balancing methods on the Cygnus load balancer to perform load shedding operations. To maximize scalability and throughput, CORBA asynchronous method invocations (AMI) [2] are used to minimize the amount of time other operations are blocked waiting for the adaptive load balancing operations to complete.

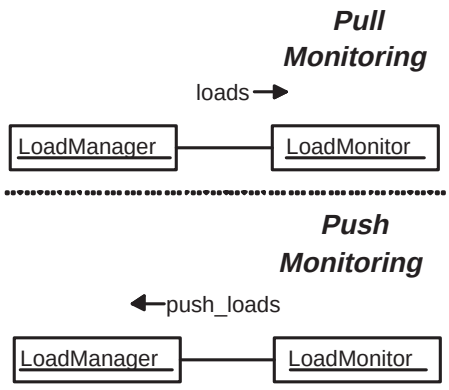


Figure 4: Load Reporting Policies

Challenge 2: Flexible Load Reporting

Context. A distributed application must be adaptively load balanced.

Problem. Adaptive load balancing requires feedback on application load conditions. Suppose the number of client requests per second is used as load metric. Request counts are typically tallied by the load balancer in a per-request architecture (see [18]), a very common load balancing architecture. However, such an architecture may not be suitable for other load metrics. Furthermore, per-request load balancing architectures incur a great deal of overhead in distributed applications. Now suppose, an on-demand architecture is used to reduce network and application overhead. Request counts can no longer be tallied by the load balancer. Furthermore, making the load balancer acquire request counts, or more generally load samples, unnecessarily restricts the types of loads that can be handled by the load balancer. These deficiencies can adversely affect the applicability of the adaptive load balancing support provided by a load balancer to a distributed application.

Solution → Load monitor. Define a load monitor component that tracks the load at a given location and reports the location load to a load balancer. As depicted in Figure 4, a load monitor can be configured with either of the following two policies:

- *Pull policy* – In this mode, a load balancer can query a given location load on-demand, *i.e.*, "pull" loads from the load monitor.
- *Push policy* – In this mode, a load monitor can "push" load reports to the load balancer.

The sole task of a load monitor component is to collect and report loads to the load balancing service. This separation of concerns greatly simplifies potential load balancing service designs and implementations, with the added benefits of improving flexibility of load reporting and reducing load sampling and reporting overhead.

Implementing the load monitor in Cygnus. Load monitors are generally application-defined objects. Consequently, Cygnus is designed to be load-metric neutral. For convenience, Cygnus is shipped with a `LoadMonitor` utility that simplifies registration of custom load monitors with its load manager. This utility also supplies a convenient means to use built-in load monitors that monitor common types of load, such as CPU load, disk load, network load, memory load, and application workload.

Challenge 3: Facilitate Transparent and Scalable Load Shedding

Context. A load balancer decides that it must shed load at given a location.

Problem. Adaptive load balancing requires the ability to shed load at a given location. It also requires a server to redirect client requests sent to its location back to the load balancer for reassignment to another location. To achieve this level of control, the load balancer must communicate with the application server(s) at a given location. However, communication with the application server(s) violates server-side transparency [17].

Solution → Load alert. Define a component that facilitates load shedding and delegate all load shedding communication to this component, rather than the application server(s). This load alert component responds to *alert* conditions set by the load analyzer component described in Challenge 1. If the load analyzer requires reduction in load (*i.e.*, it must shed load) from an object group member location, it enables an "alert" condition on the load alert component residing at that same location. After the alert is enabled, the load alert component rejects client requests. Requests are rejected by a server request interceptor that throws a `CORBA::TRANSIENT` exception. When a client ORB receives that exception, it will transparently reissue the request to the original target, *i.e.*, the load

balancer. The load balancer will then transparently reassign the client's request to another member in the object group.

Implementing load alerts in Cygnus. Applications may register load alert objects with Cygnus. Cygnus maps load alert objects to object group members using an efficient hash map. This design minimizes load alert object lookup, which enhances the overall scalability of Cygnus itself.

Cygnus invokes the application-defined load alert objects to enable or disable load shedding on a given object group member. It uses AMI to improve overall throughput in Cygnus, as outlined in Challenge 1. The use of AMI reduces the overhead of Cygnus by minimizing blocking time.

A load alert object consists of (1) a servant that the load balancer can invoke requests on and (2) a server request interceptor that performs the actual load shedding by intercepting client requests and determining whether or not they should be rejected. The amount of overhead incurred by the interception of client requests depends largely on the efficiency of TAO's Portable Interceptor[3] implementation. For example, when an alert is not enabled an interception can be reduced to an instantiation of a small object and a simple atomic boolean flag check.

2.3 Dynamic Interactions in the Proposed OMG Load Balancing and Monitoring Service

Section 2.2 describes the static relationships among the components in Cygnus. This section augments this discussion by describing the dynamic interactions among these components. Although the following discussion is not comprehensive, the scenario focuses on the case where the location an object group member resides at has become overloaded, causing requests to be redirected. This scenario was chosen since it illustrates all interactions that occur between a client, adaptive load balancing service, and a group of objects or servers comprising an object group.[4]

Selecting a target member using a non-adaptive balancing policy can yield non-uniform loads across group members. In contrast, selecting a member adaptively for each request can incur excessive overhead and latency. To avoid either extreme, Cygnus therefore provides a hybrid solution [18], whose interactions are shown in Figure 5. Each interaction in Figure 5 is outlined below.

[3] A Portable Interceptor is an instance of the Interceptor design pattern [22], with an interface defined by the OMG, designed to be registered with an application's ORB and invoked at various request processing points with the intention of either examining the contents of the request or preventing the request from continuing.

[4] Since the non-adaptive case is a subset of the adaptive case, we omit such scenarios, such as the interactions that occur between a client, a *non*-adaptive load balancing service, and group of objects or servers.

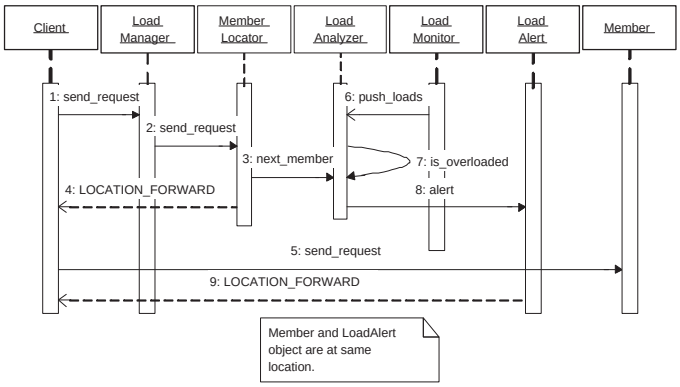


Figure 5: Cygnus Load Balancing and Monitoring Interactions

1. A client obtains an object reference to what it believes to be a CORBA object and invokes an operation. In actuality, however, the client transparently invokes the request on the load manager itself.
2. After the request is received from the client, the load manager's POA dispatches the request to its servant locator, *i.e.*, the member locator component.
3. Next, the member locator queries the load analyzer for an appropriate group member.
4. The member locator then transparently redirects the client to the chosen member.
5. Requests will continue to be sent *directly* to the chosen member until the load analyzer detects a high load at the location the member resides. The additional indirection and overhead incurred by per-request load balancing architectures (see [18]) is eliminated since the client communicates with the member directly.
6. The load monitor monitors a location's load. Depending on the load reporting policy (see *load monitor* description in Section 2.2) that is configured, the load monitor will either report the load(s) to the load analyzer (via the load manager) or the load manager will query the load monitor for the load(s) at a given location.
7. As loads are collected by the load manager, the load analyzer analyzes the load at all known locations.
8. To fulfill the transparency requirements, the load manager does not communicate with the client application when forwarding it to another member after it has been bound to a member. Instead, the load manager issues an "alert" to the `LoadAlert` object residing at the location the member resides at. Depending on the contents of the alert issued by the load manager, the `LoadAlert` object will either cause request be accepted or redirected.
9. When instructed by the load analyzer, the `LoadAlert` object uses the GIOP LOCATION_FORWARD message to

dynamically and transparently redirect subsequent requests sent by one or more clients back to the load manager.

After all these steps, the load balancing cycle begins again. Note that this hybrid approach does not *not* perform load balancing on a per-request basis. It performs load balancing on-demand, thus avoiding a major bottleneck found in many other load balancing implementations.

3 Empirical Results

To improve overall application performance significantly, a load balancing service itself must incur minimal overhead. A key contribution of the Cygnus load balancing and monitoring (LB/M) service described in Section 2.2 is its ability to increase overall system scalability. The Cygnus LB/M service achieves scalability by distributing requests across multiple back-end servers (object group members). It is also designed to avoid increasing round-trip latency and jitter significantly.

This section describes the design and results of several experiments performed to empirically quantify the benefits of the Cygnus adaptive on-demand load balancing support, as well as to demonstrate the limitations with the alternative load balancing strategies outlined in [18]. Section 3.1 outlines the hardware and software platform used to benchmark Cygnus. Section 3.2 presents the results from a set of experiments that illustrate the improved scalability attained by introducing Cygnus' adaptive load balancing capabilities into a representative distributed application.

3.1 Hardware/Software Benchmarking Platform

Benchmarks performed for this paper were run on Emulab[5] using between 2 and 49 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 6. All benchmarks were run in the POSIX real-time thread scheduling class [12]. This scheduling class enhances the consistency 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 single-threaded form of the "`Latency`" performance test distributed with the TAO open-source software release.[6] Only stateless objects are used as targets in this test. All benchmarks were configured to run 200,000 iterations and to generate the same load. The figure in Sidebar 1 illustrates the basic design of this performance test. All benchmarks use one of the following variations of the `Latency` test:

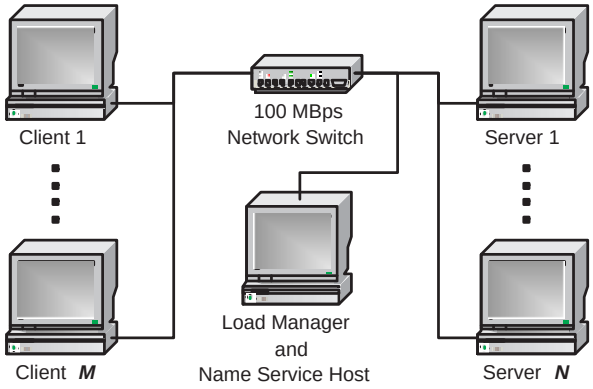


Figure 6: Load Balancing Experiment Testbed

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. As defined, by the proposed CORBA LB/M specification, the Round Robin 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 rotating through the list of members in a given object group. 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 non-adaptive load balancing and selects a member at random from the list of members in a given object group.
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 load balancing strategy. As its name implies, it chooses the object group member with the lowest load, which is computed dynamically.

3.2 Scalability Results

The primary use of a load balancer is to improve scalability. As such, it is important to demonstrate that a particular load

[5] Emulab (`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.

[6] `TAO_ROOT/performance-tests/Latency/Single_Threaded` in the TAO release contains the source code for this benchmark.

balancer configuration actually improves distributed application scalability. Three sets of benchmarks are shown below, one for each load balancing strategy defined by the proposed CORBA LB/M specification: (1) Round Robin, (2) Random, and (3) Least Loaded. Each set of benchmarks shows how throughput and latency vary as the number of clients is increased between 1 and 16 clients, and the number of servers is increased between 1 and 16 servers. In general, only two or three server data sets are shown to illustrate trends without cluttering the benchmark graphs.

3.2.1 Round Robin Strategy Benchmarks

Figure 7 shows how client request throughput varies as the number of clients and servers are increased when using the Round Robin load balancing strategy. This figure shows how

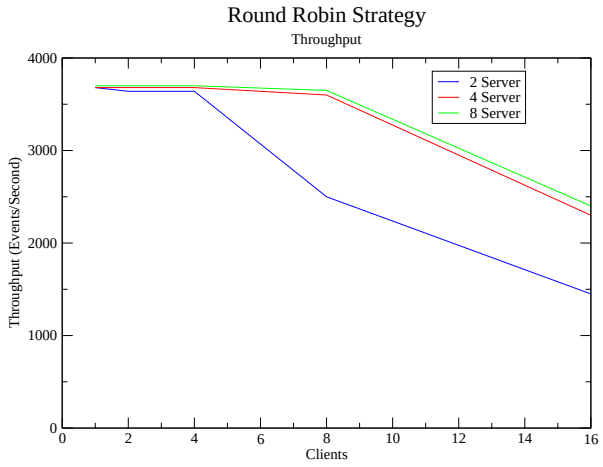


Figure 7: Round Robin Strategy Throughput

throughput decreased as the number of clients were increased beyond the same number of servers. For example, throughput remained essentially unchanged as long as the number of clients was less than the number of servers. These results demonstrate that the Round Robin load balancing strategy implemented by Cygnus incurs no overhead beyond the first request. In particular, Cygnus no longer participates in subsequent client requests after it binds a client to a given object group member via the Round Robin load balancing strategy.

Figure 7 also shows that as the number of servers increased, throughput also increased when the number of clients surpassed the number of servers. For example, when the number of clients is 8, the throughput with 4 servers is more than the throughput with 2 servers.

Figure 8 illustrates how request latency varied as the number of clients and servers were increased. This figure shows how employing Cygnus in the `Latency` performance test improved both throughput and latency. Increasing the number of

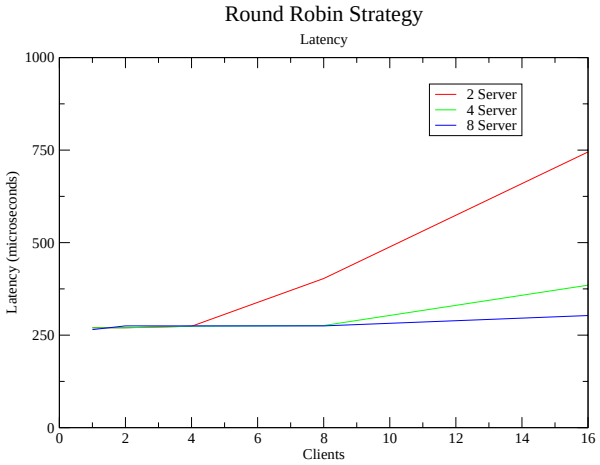


Figure 8: Round Robin Strategy Latency

servers improved the latency. For example, the latency for the 16 client and 2 server case is approximately 750 microseconds. Increasing the number of servers to 8 reduced the roundtrip latency to about 300 microseconds. This decrease in latency in turn increased the throughput as the number of servers increased.

3.2.2 Random Strategy Benchmarks

Figure 9 depicts how the Random load balancing strategy implemented in Cygnus behaved when varying the number of clients and servers. This figure shows how the throughput

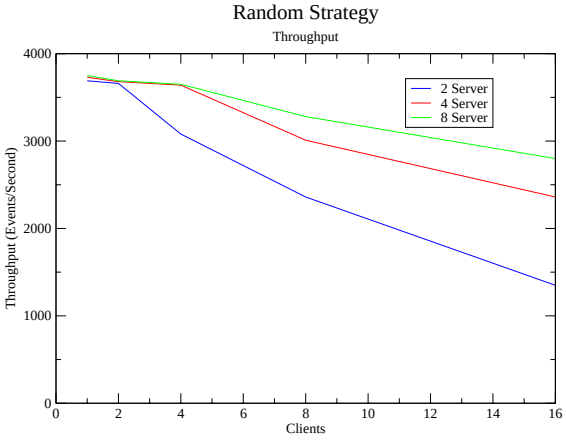


Figure 9: Random Strategy Throughput

of the Random load balancing strategy behaved basically the same as the Round Robin load balancing strategy presented in Section 3.2.1. Both strategies exhibit similar overhead and scalability characteristics due to the fact that they are non-adaptive and have fairly simple member selection algorithms.

The results in Figure 9 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.

Figure 10 shows roundtrip latency for the Random load balancing strategy case increases when the number of clients exceeds the number of servers. This behavior occurs because the random strategy continues to bind certain client requests to the same server, even though other less loaded servers are available. As shown in this figure, latency improved (*i.e.*, decreased) as the number of servers increased.

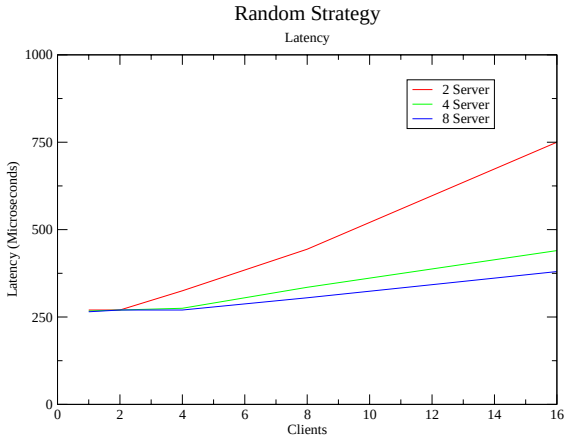


Figure 10: Random Strategy Latency

3.2.3 Least Loaded Strategy Benchmarks

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

- A load monitor process that measured the number of requests per second and residing within the server was registered with the Cygnus
- A *reject threshold* of 10,000 events/second was set, which is the threshold at which Cygnus will avoid selecting a member with that load.
- A *critical threshold* of 30,000 events/second was set, which is the threshold at which Cygnus informs servers to shed loads by redirecting requests back to Cygnus.
- A *dampening* of 0.2 was set, which is the value that determines what fraction of a newly reported load is considered when making load balancing decisions.

With this configuration, Cygnus queried the server load monitor every 5 seconds (the Cygnus default). Moreover, high load conditions caused Cygnus to either reject object group members when selecting members to bind request to, or caused Cygnus to request that servers shed load.

Figure 11 illustrates how Cygnus' Least Loaded load balancing strategy reacts as the number of clients and servers increased. This figure illustrates how Cygnus' Least Loaded strategy incurs certain overhead compared with the Round Robin and Random strategies. This overhead included (1) taking into account member loads, (2) rejecting some members during member selection, and (3) shedding loads when some servers become overloaded.

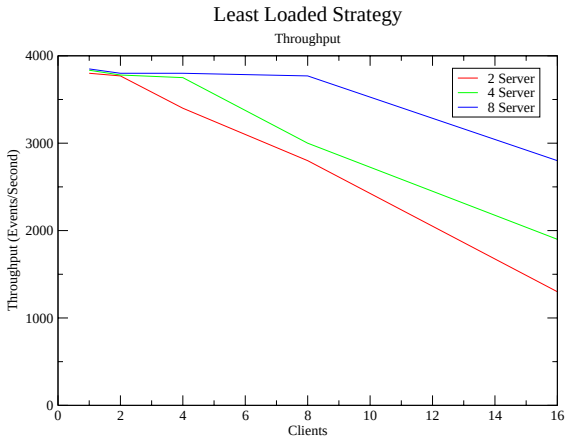


Figure 11: Least Loaded Strategy Throughput

- 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 11 illustrates that scalability still improved. In particular, increasing the number of servers showed further improvements in scalability. When the number of clients is 8 and the number of servers is 2 there is a good difference between the throughput obtained from the Random strategy and the throughput obtained from Least-Loaded strategy. This behavior occurs because the Random strategy tries to bind the client to the same server even though other less loaded servers are available. In contrast, Cygnus' adaptive load balancer balances the load accordingly, which therefore increases throughput.

The latency results shown in Figure 12 illustrate reductions in roundtrip latency as the number of servers are increased.

There are certain cases when the latency is more than the latency for the Round Robin and the Random strategy cases. These results can also be attributed to the additional overhead caused by Cygnus' adaptive load balancing mechanisms.

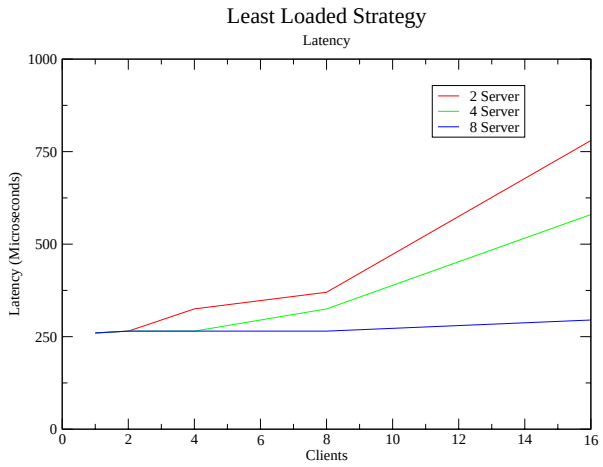


Figure 12: Least Loaded Strategy Latency

3.3 Summary of Results

Section 3.2 showed that the proposed 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 added in the adaptive load balancer does not add any overhead compared with the non-adaptive Round Robin and the Random strategies.

Given a test configuration with clients generating non-uniform loads, the benefits of adaptive load balancing would be more evident. However, results and discussions showing these benefits are beyond the scope of this paper (see our prior work on adaptive load balancing [18, 17] for concrete results based on an earlier prototype of Cygnus). 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 application scalability. As our results show, scalability was indeed improved in all test cases.

4 Related Work

This section compares and contrasts our work on middleware load balancing and Cygnus with representative related work. Middleware load balancing provides the most flexibility in terms of influencing how a load balancing service makes decisions, and in terms of applicability to different types of distributed applications [6, 8]. Load balancing at this level, as depicted in Figure 13, provides for straightforward selection of load metrics, in addition to the ability to make load balancing decisions based on the content of a request.

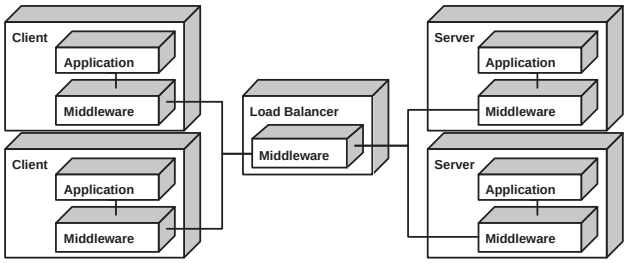


Figure 13: Middleware-level Load Balancing

Some middleware implementations integrate load balancing functionality into the ORB middleware [13] itself, whereas others implement load balancing support at the service level. The latter is the approach taken by Cygnus upon which the content of this paper is based.

CORBA load balancing. An increasing number of projects are focusing on CORBA load balancing, which can be implemented at the following levels in the OMG reference architecture.

• **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 [13]. For example, Inprise's VisiBroker implements a similar strategy, where Visibroker's object adapter [10] 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. The Cygnus load balancing service therefore does not rely on

ORB-level extensions or non-standard features, *e.g.*, 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. Like the Visibroker implementation and the strategies described in [13], Cygnus' approach is transparent to clients. Unlike the ORB-based approaches, however, Cygnus only uses standard CORBA features. It can therefore 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 [9] 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 [11] 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 [18]. BEA's WebLogic [4] uses a per-request load balancing strategy, also described in [18]. 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 Orbix's load balancing service.

5 Concluding Remarks

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

Load can be balanced at several layers, including the network, OS, middleware, and application. Network-level and OS-level load balancing architectures are generally inflexible since they cannot support *application-defined* metrics at runtime when making load balancing decisions. They also lack adaptability due to the absence of load-related feedback from a given set of object group members, as well as the inability to control if and when a given member should accept additional requests. Likewise, application-level load balancing suffers from lack of transparency, increased code complexity, and increased maintenance burden.

To address these limitations, we have devised an adaptive middleware load balancing architecture – called Cygnus – to overcome the limitations with network-based and OS-based load balancing mechanisms outlined above. This paper motivates and describes the design and performance of Cygnus, which is an implementation of a CORBA Load Balancing and Monitoring (LB/M) service proposal developed using the standard CORBA features provided by the TAO ORB [21].

The results in this paper illustrate 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. For example, the empirical results in Section 3 show that introducing LB/M into distributed applications can substantially improve scalability with minimal run-time overhead. As a result, developers can concentrate on their core application behavior, rather than wrestling with complex infrastructure mechanisms needed to make their application distributed and scalable.

The Cygnus LB/M service implementation is based entirely on standard CORBA features, such as location forwarding, servant locators and asynchronous method invocation (AMI), which demonstrates that CORBA technology has matured to the point where many higher-level services can be implemented efficiently without requiring extensions to the ORB or its communication protocols. Exploiting the rich set of primitives available in CORBA still requires specialized skills, however, along with the use of somewhat poorly documented features. Further research and documentation of the effective architectures and patterns used in the implementation of higher-level CORBA services is therefore needed to advance the state of the practice and to allow application developers to make better decisions when designing their systems.

TAO and Cygnus have been applied to a wide range of distributed applications domains. Chief among these domains include telecommunications, aerospace, defense, online financial trading, medical, and manufacturing process control. PrismTechnologies has developed a Java implementation of the proposed OMG CORBA LB/M that interoperates with the Cygnus C++ implementation provided with TAO.

References

[1] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX Summer Conference*. USENIX, June 2000.

[2] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In

Proceedings of the Middleware 2000 Conference. ACM/IFIP, April 2000.

[3] Seán Baker. *CORBA Distributed Objects using Orbix*. Addison Wesley, 1997.

[4] BEA Systems Inc. WebLogic Administration Guide. edoc.bea.com/wle/.

[5] Cisco Systems, Inc. High availability web services. www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/mnibm_wp.htm, 2000.

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

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] 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.

[9] 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'99)*, Antwerp, Belgium, September 2000. OMG.

[10] 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.

[11] IONA Technologies. Orbix 2000. http://www.iona.com/products/orbix2000_home.htm.

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

[13] Markus 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, September 2000. OMG.

[14] Object Management Group. *CORBA Components*, OMG Document formal/2001-11-03 edition, November 2001.

[15] Object Management Group. *Proposed CORBA Load Balancing and Monitoring Specification*, OMG Document mars/02-10-14 edition, October 2002.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0 edition, June 2002.

[17] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. Designing an Adaptive CORBA Load Balancing Service Using TAO. *IEEE Distributed Systems Online*, 2(4), April 2001.

[18] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. Strategies for CORBA Middleware-Based Load Balancing. *IEEE Distributed Systems Online*, 2(3), March 2001.

[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] Richard E. Schantz and Douglas C. Schmidt. Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications. In John Marciniak and George Telecki, editors, *Encyclopedia of Software Engineering*. Wiley & Sons, New York, 2002.

[21] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.

[22] 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.

[23] Sun Microsystems, Inc. *Java Remote Method Invocation Specification (RMI)*, October 1998.