Evaluating the Performance of Middleware Load Balancing Strategies

Jaiganesh Balasubramanian, Douglas C. Schmidt, Lawrence Dowdy, and Ossama Othman
{jai,schmidt,ossama}@dre.vanderbilt.edu and larry.dowdy@vanderbilt.edu
Institute for Software and Integrated Systems
Vanderbilt University, Nashville, TN, USA

Abstract

This paper presents three contributions to research on middleware load balancing. First, it describes the design of Cygnus, which is an extensible open-source middleware framework developed to support adaptive and non-adaptive load balancing strategies. Key features of Cygnus are its ability to make load balancing decisions based on application-defined load metrics, dynamically (re)configure load balancing strategies at run-time, and transparently add load balancing support to client and server applications. Second, it describes the design of LBPerf, an open-source middleware load balancing benchmarking toolkit developed to evaluate load balancing strategies at the middleware level. Third, it presents the results of experiments that systematically evaluate the performance of adaptive load balancing strategies implemented using the Cygnus middleware framework using workloads generated by LBPerf. The workloads used in our experiments are based on models of CPU-bound requests that are representative of a broad range of distributed applications.

Our experiments with LBPerf illustrate the need for evaluating different adaptive and non-adaptive load balancing strategies under different workload conditions. In addition to assisting in choosing a suitable load balancing strategy for a particular class of distributed applications, our empirical results help configure run-time parameters properly and analyze their behavior in the presence of different workloads. Our results also indicate that integrating Cygnus into distributed applications can improve their 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 enhance the scalability of their distributed applications.

Keywords: Middleware load balancing, adaptive load balancing strategies.

1 Introduction

Load balancing is a well-established technique for utilizing available computing resources more effectively by partitioning tasks according to load distribution strategies. Distributed applications can improve their scalability by employing load balancing in various ways and at various system levels. For example, heavily accessed Internet web sites often use load balancing at the network [4] and operating system [13] 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 applications (such as online stock trading, weather prediction, and total ship computing environments [20]) due to the lack of application-level control over load balancing policies, lack of extensible load metrics, and inability to know the distributed system state/behavior and client request content when distributing loads.

For these types of distributed applications, load balancing at the *middleware* level can help improve scalability without incurring the limitations of load balancing at lower levels outlined above. As with load balancing done at other levels, middleware load balancing can be *adaptive* or *non-adaptive*, depending on whether or not run-time load conditions influence load balancing decisions. Adaptive load balancing strategies use run-time system state information (*e.g.*, host workload), to make load balancing decisions, whereas non-adaptive load balancing strategies do not.

In theory, adaptive load balancing can be more flexible and desirable than non-adaptive load balancing since it can satisfy key distributed application requirements, such as:

- Improved handling of erratic client request patterns
- Optimizing resource utilization for different workloads, workload conditions, and types of systems.

In practice, however, the effectiveness of adaptive load balancing depends on the load distribution strategy chosen, the type of load metrics chosen, and on other run-time parameters needed by the adaptive strategies. For example, an influential run-time parameter is the *load metric* [14] a load balancer uses to make load balancing decisions. Since load balancing middleware can be employed using various load balancing strategies and multiple metrics, determining the appropriate adaptive load distribution strategies for different classes of distributed applications is hard without the guidance of comprehensive performance evaluation models, systematic benchmarking metrics, and detailed empirical results.

Our earlier work [18, 17, 19] on middleware load balancing focused on (1) defining a nomenclature that can describe various forms of load balancing middleware, (2) creating a flexible and portable load balancing model, (3) identifying advanced features of load balancing middleware that can be used to enhance and optimize this load balancing model, (4) designing a middleware framework that supports the RoundRobin, Random, and LeastLoaded load balancing strategies, and (5) developing an efficient load balancing service that implements the strategies using standard middleware features.

This paper explores a previously unexamined topic pertaining to load balancing middleware by *characterizing workload models in distributed systems and empirically evaluating the suitability of alternative middleware load balancing strategies for distributed applications that generate these workloads*. We evaluate these strategies in the context of two technologies:

- *LBPerf*, which is a benchmarking toolkit for evaluating middleware load balancing strategies.
- *Cygnus*, which is an extensible C++ middleware framework that supports a range of adaptive and non-adaptive load balancing strategies.

LBPerf and Cygnus are open-source and available from `deuce.doc.wustl.edu/Download.html`.

We developed Cygnus to investigate the pros and cons of adaptive and non-adaptive load balancing strategies operating in standard middleware distributed systems. The results of applying LBPerf to Cygnus show that well-designed adaptive load balancing strategies can improve the performance of the system in the presence of a range of CPU loads. Moreover, the threshold values associated with adaptive load balancing strategies help meet key quality of service (QoS) requirements needed by certain types of distributed applications.

The remainder of this paper is organized as follows: Section 2 presents our load balancing and distributed system models; Section 3 examines the architecture of the Cygnus middleware framework used to implement the load balancing experiments conducted in this paper; Section 4 introduces the LBPerf middleware load balancing toolkit used to drive the benchmarking experiments in this paper and analyzes empirical results that evaluate the suitability of various middleware load balancing strategies for distributed applications; Section 5 compares our research on middleware load balancing with related work; and Section 6 presents concluding remarks.

2 Middleware Load Balancing Architecture and Distributed System Model

This section presents the architecture our middleware load balancing service and distributed system model. We first illustrate the key requirements for a middleware load balancing service and provide an overview of the architectural concepts and components of our middleware load balancing service. We then describe the distributed system model used in the evaluation of this service.

2.1 Key Middleware Load Balancing Requirements

The following are the key requirements of a middleware load balancing service [18]:

- **General purpose** – A load balancing service should make little or no assumptions about the types of applications whose loads it balances.
- **Transparency** – A load balancing service should balance loads in a manner transparent to client applications (and as transparently as possible to servers).
- **Adaptive** – A load balancing service should be able to adapt its load balancing decisions based on dynamic load changes.
- **Scalable and Extensible** – A load balancing service should provide scalability to a distributed application by utilizing available computing resources to handle a large number of client requests and manage many servers efficiently, and should be neutral to different load balancing strategies.

2.2 The Structure and Dynamics of a Middleware Load Balancing Architecture

The key middleware load balancing concepts we defined to meet the requirements outlined in Section 2.1 are shown in Figure 1 and described below:

- **Load balancer**, which is a component that attempts to distribute the workload across groups of servers in an optimal manner. 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 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.*, is *stateful*) or retain no state at all (*i.e.*, is *stateless*).
- **Object group**, which is a group of *members* across which loads are balanced. Members in such groups implement the same remote operations.
- **Session**, which in the context of load balancing middleware defines the period of time that a client invokes remote operations to access services provided by objects in a particular server.

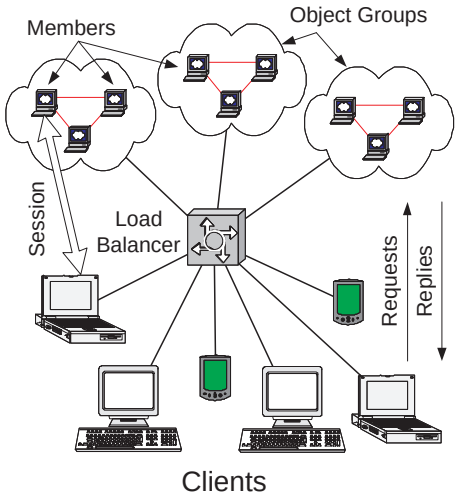

Figure 1. Key Middleware Load Balancing Service Concepts

2.3 Distributed System Model

Figure 2 shows the model of the distributed system used as the basis for the load balancing experiments in this paper. In this model, distributed systems are built by interconnecting clients and servers. Each server is a member of an object group. All servers register themselves with a load balancer, which itself consists of one or more servers that mediate between different elements of the load balancing service.

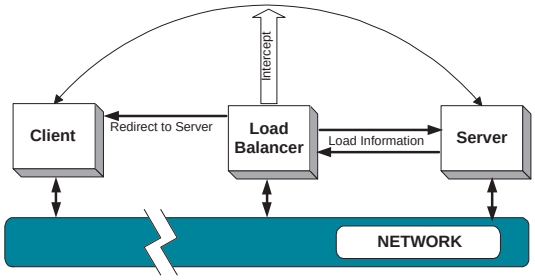

Figure 2. Distributed System Model

When a client invokes an operation, this request is initially intercepted by the load balancer. The load balancer finds an object group member (*i.e.*, a server) suitable to handle the request and then redirects the operation request to the designated group member using the underlying middleware request redirection support. Client applications are thus assigned a group member to handle their request. The overhead of contacting the load balancer is incurred only for the *first* request. Henceforth, the client communicates to its designated server directly, unless further dynamic adaptation is needed (and supported by the load balancing service, of course).

As with the load balancing service described in Section 2.2, the distributed system model presented above can be implemented on a range of middleware platforms. Section 3 describes how we implemented this model using the Cygnus load balancing middleware framework.

3 Overview of Cygnus

This section presents an overview of the Cygnus load balancing framework, which is a CORBA [16] implementation of the middleware load balancing architecture described in Section 2.2. In addition to explaining the responsibilities of the key components in Cygnus, we describe Cygnus's load balancing strategies and load monitors and show how all these elements interact at run-time. Section 4 then describes benchmarks that evaluate the performance of Cygnus empirically.

3.1 Components in the Cygnus Load Balancing Framework

Figure 3 illustrates the components in the Cygnus load balancing framework. The components in this figure form a middleware load balancing framework that provides the following capabilities:

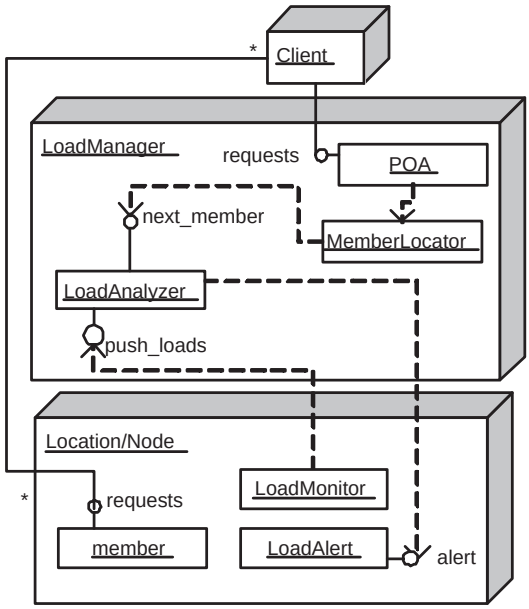

Figure 3. Load Balancing Components in Cygnus

- The **load manager** is a *mediator* [8], that is the application entry point for all load balancing tasks. This component facilitates client and object group member

participation in load balancing decisions without exposing them to implementation details that should remain internal to the load balancing service.

- The **member locator** is the *interceptor* [21] that binds a client to an object group member. This component allows the load balancer to transparently inform the client to invoke requests on the chosen object group member.
- The **load analyzer** examines load conditions and triggers load shedding when necessary based on the load balancing *strategy* in use. This component also decides which object group member will receive the next request. Distributed applications whose loads change frequently can use the load analyzer to self-adaptively choose new load metrics and load balancing strategies at run-time.
- The **load monitor** makes load reports available to the load manager, thereby decoupling potentially application-specific load retrieval tasks from the load balancer. This component also allows the load balancer to independent of the load-metric.
- The **load alert** acts as a *mediator* between the load balancer and the object group member. It shields the object group member from load shedding instructions issued by the load balancer by handling those instructions and taking appropriate actions, such as intercepting client requests and transparently redirecting them to the load balancer.

3.2 Cygnus Load Balancing Strategies

The following load balancing strategies are currently integrated into the Cygnus middleware framework described in Section 3.1:

- **RoundRobin** – This non-adaptive strategy keeps a list of locations containing at least one object group member, and selects members by simply iterating through that location list.
- **Random** – This non-adaptive strategy keeps a list of locations where object group members are present and randomly chooses a location to serve a request.
- **LeastLoaded** – This adaptive strategy allows locations to continue receiving requests until a threshold value is reached. Once the threshold value is reached, subsequent requests are transferred to the location with the lowest load.
- **LoadMinimum** – This adaptive strategy calculates the average loads at locations containing object group members. If the load at a particular location is higher than the average load and greater than the least loaded location by a certain *migration threshold percentage*, all subsequent requests will be transferred to the least loaded location.

Two key elements of Cygnus's adaptive load balancing strategies are their *transfer* and *granularity* policies [6]. The transfer policy determines whether a request should continue to be processed at the server where it is currently being processed or migrated to another server. The granularity policy determines which object group member receives the request selected for execution.

Transfer policies typically use some type of *load metric* threshold to determine whether the particular location is overloaded. Cygnus supports the following load metrics:

- **Requests-per-second**, which calculates the average number of requests per second arriving at each server.
- **CPU run queue length**, which returns the load as the average number of processes in the OS run queue over a specific time period in seconds, normalized over the number of processors.
- **CPU utilization**, which returns the load as the CPU usage percentage.

Cygnus load monitor components outlined in Section 3.1 measure the loads at each endsystem based on the configured load metrics.

Granularity policies typically focus on entities such as a host or process. For example, a host-oriented load balancer would balance loads across multiple hosts and a process-oriented load balancer would balance loads across multiple processes. Since certain granularity levels may not be suitable for all types of applications, Cygnus performs load balancing at a more abstract unit of granularity, namely *location*, which can be defined to be whatever the application developer chooses. For instance, members of one object group could be load balanced at the host level, whereas members of another object group could be balanced at the process level. Cygnus only sees abstract locations, however, which increases its flexibility since its load balancing strategies need not be designed for specific granularities.

3.3 Dynamic Interactions in the Cygnus Load Balancing Framework

Figure 4 illustrates how Cygnus components (Section 3.1) interact with each other and with the load balancing strategies (Section 3.2) at run-time. As shown in this figure, the following interactions occur when Cygnus components process client requests:

1. A client invokes an operation on what it believes to be its target CORBA object. In actuality, however, the client transparently invokes the request on the load manager itself.
2. The load manager dispatches that request to its member locator component.
3. The member locator component queries the load analyzer component for an object group member (server) to handle the request.

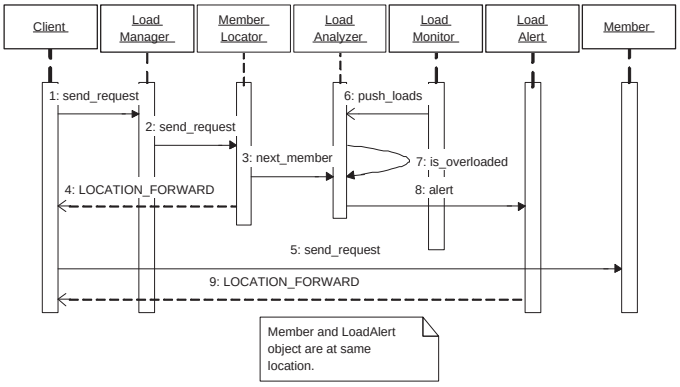

Figure 4. Interactions Between Cygnus Components

4. After a member is chosen by the load analyzer, the member locator component uses the standard CORBA GIOP LOCATION_FORWARD message to dynamically and transparently redirect the client to that member.
5. Clients start sending requests directly to the chosen object group member. The load manager component is not involved in any requests that are sent, eliminating the additional indirection and overhead incurred by per-request load balancing architectures [18].

   The remaining activities described below are carried out by Cygnus only when adaptive load balancing strategies are used.
6. The load monitor monitors a location's load and this information is reported to the load analyzer.
7. As loads are collected, the load analyzer analyzes the load at all known locations.
8. When the load analyzer detects that an object group member is overloaded, the load manager does not inform the client that it needs to contact another less loaded object group member. To fulfill the transparency requirements, the load manager issues an *alert* to the load alert component residing at the location where the object group member resides.
9. When instructed by the load analyzer, the load alert component uses the GIOP LOCATION_FORWARD message to dynamically and transparently redirect subsequent requests sent by one or more clients back to the load manager, whose object reference it obtained from the contents of the alert message.

4 Empirical Results

This section describes the design and results of experiments that empirically evaluate the performance of Cygnus's adaptive and non-adaptive load balancing strategies outlined in Section 3.2. These results illustrate how Cygnus addresses the key middleware load balancing requirements outlined in Section 2.1. In particular, they show:

- How Cygnus improves the scalability of distributed applications, without incurring significant run-time overhead.
- How well Cygnus's adaptive load balancing strategies adapt to changing workload conditions.
- How different threshold values help maintain QoS properties needed by certain distributed applications.

We start by describing the workload model used to evaluate the performance of Cygnus's load balancing strategies empirically. We next describe the hardware and software infrastructure used in our experiments. We then describe the experiments themselves, focusing on the load metrics and the run-time configuration for the adaptive load balancing strategies we evaluate. Finally, we present and analyze the empirical results.

4.1 Workload Model

A key motivation for studying load balancing performance is to compare and contrast different architectures and strategies. The performance of these architectures and strategies is often evaluated by conducting benchmarking experiments under different environments and test configurations [23]. The results of such experiments are not particularly useful, however, unless the system workload is representative of the context in which the load balancer and applications will actually be deployed. This section therefore describes the workload model we use in Section 4.5 to empirically evaluate the performance of Cygnus's load balancing strategies described in Section 3.2.

4.1.1 Workload Characterization for Load Balancing

Accurately characterizing workload is important when planning and provisioning scalable distributed systems. This activity starts by defining a distributed system model (see Section 2.3) that conveys the components in the system, how these components interact, and what types of inputs are possible. This activity produces a *workload model*, which could be:

- **A closed analytical queuing network model** [5], where workloads are characterized by information such as input parameters, job arrival rates, and system utilization.
- **A simulation model** [22], where workloads include additional scenarios, such as concurrency and synchronization.

- **An executable model** [23], where workloads capture the request traffic patterns experienced by the system to help demonstrate empirically how the system will behave.

This paper focuses on executable workload models because our performance studies are based on the Cygnus middleware load balancing framework described in Section 3. We are therefore interested in analyzing the behavior of Cygnus using workloads that are characteristic of actual application scenarios. To facilitate this analysis, we developed a benchmarking toolkit called LBPerf, which generates workloads that mimic the processing performed in middleware-based distributed systems and then uses these workloads to test the behavior of middleware load balancing services. Based on the the model described in Section 2.3, workloads generated by LBPerf can be further classified according to:

- **Resource type**, where workloads can be characterized by the type of resource being consumed, *e.g.*, CPU usage, I/O usage, and overall system usage.
- **Service type**, where workloads can be characterized by the type of service performed by servers on behalf of clients, *e.g.*, a database transaction, an e-mail transaction, or web-based search.
- **Session type**, where workloads can be characterized by the type of requests initiated by a client and serviced by a server in the context of a session. These requests can be of any type, including the resource type and the service type of requests.

4.1.2 Workload Characterization for Cygnus

To compare the performance of the adaptive load balancing strategies (Section 3.2) implemented in the Cygnus middleware load balancing framework (Section 3.1), this paper focuses on **session type** workloads for middleware-based distributed systems.

We do not consider **resource type** workloads since in middleware-based distributed systems it is possible to receive simple requests (*e.g.*, to obtain the server's object reference from a Naming or Directory Service) that do not incur significant resource usage. So the middleware load balancing service should not be tied down to the resource usage of requests and should be usage neutral.

We also do not consider **service type** workloads because of the different resource usage associated with those workloads. For example, a single request could fetch a large table of numbers from a database (resulting in I/O usage) and then determine if any number is prime (resulting in CPU utilization). Designing a single load metric to measure the load in such cases can be very hard. We therefore focus on **session type** workloads, which include both resource and service type workloads, *i.e.*, the resource usage associated with those workloads can be either CPU or I/O. This paper, however, just focuses on session type workloads that incur CPU resource usage.

Section 4.5 describes results from empirical experiments that apply the workload model presented above to evaluate the performance of Cygnus's load balancing strategies. These experiments make the following assumptions pertaining to the workload model discussed above:

- Load balancers used the run-time load information, but did not use the type of their client requests to make decisions on which server will run the requests.
- Though Cygnus need not make any assumptions about the servers whose loads it balances, all server hosts in the experiments we ran were homogeneous, *i.e.*, they all had similar processing capabilities.
- Session length was not known *a priori*, *i.e.*, the load balancer and servers did not know in advance how much time was spent processing client requests.

4.2 Hardware/Software Testbed

All benchmarks conducted for this paper were run on Emulab (`www.emulab.net`), which is an NSF-sponsored testbed at the University of Utah that facilitates simulation and emulation of different network topologies for use in experiments that require many nodes. As shown in Figure 5, we used Emulab to emulate the distributed system model described in Section 2.3. Our Emulab experiments

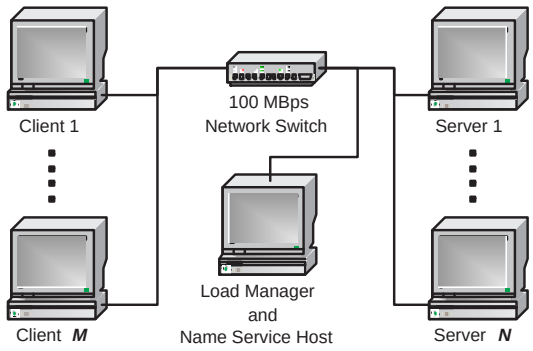

Figure 5. Load Balancing Experiment Testbed

used between 2 and 41 single CPU Intel Pentium III 850 MHz and 600 MHZ PCs, all running the RedHat Linux 7.1 distribution, which supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. All PCs were connected over a 100 Mbps LAN and version 1.4.1 of TAO was used. The benchmarks ran in the POSIX real-time thread scheduling class [12] to enhance the consistency of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

4.3 Benchmarking Experiments and Results

The core CORBA benchmarking software is based on the single-threaded version of the "`Latency`" performance test distributed with the LBPerf open-source benchmarking toolkit. The `Latency` performance test creates a session for clients to communicate with servers by making a configurable number of iterated requests. All benchmarks tests can be configured with the load balancing strategies supported by Cygnus, which currently include the RoundRobin, Random, LeastLoaded, and LoadMinimum strategies described in Section 3.2. LBPerf can conduct the following measurements for a session between the clients and the servers:

- **Throughput**, which is the number of requests processed per second by the server during a particular session with the client.
- **Latency**, which is the roundtrip time taken for a particular request from the time it was invoked by the client until the reply for the request came from the server.

To emulate the workload model described in Section 4.1, LBPerf enables developers to generate whatever workload is representative of particular distributed applications whose workloads require balancing.

4.3.1 Run-time Configuration of Adaptive Load Balancing Strategies

Prior work [6, 14] shows the performance of adaptive load balancing strategies depends on the following factors:

- The load metric(s) used to make migration and balancing decisions. Common metrics include requests-per-second, CPU run-queue length over a given period of time, and CPU utilization.
- The parameters chosen to configure the adaptive load balancing strategies at run-time. Common parameters include the interval chosen for reporting server loads and threshold values that affect load balancing decisions when analyzing those loads.

Section 3.2 described the load metrics supported by Cygnus. We now describe how the benchmarking experiments conducted for this paper used the Cygnus run-time configuration parameters described below.

- **Load reporting interval**, which determines the frequency at which server loads are propagated to the load balancer. The *load reporting interval* should be chosen depending on the current load and request arrival rate in the system [22]. Since the benchmarking experiments in Section 4.5 cover different workload scenarios, the load reporting interval used is stated for each experiment.
- **Reject and critical threshold values**, which can be set for the LeastLoaded load balancing strategy. The *reject threshold* determines the point at which Cygnus will avoid selecting a member with a load greater than or equal to that load threshold. The *critical threshold* determines the point at which Cygnus informs servers to shed loads by redirecting requests back to Cygnus.

  Changing threshold values to correspond to the workload can improve client response time [23]. For example, if the number of requests is very high and the load metric is the number of requests-per-second at each location, the reject and critical threshold values should be higher than when the number of requests is comparatively lower. Otherwise, incoming requests will frequently exceed the reject and critical threshold values and trigger unnecessary load migrations and unstable load balancing decisions. These threshold values must therefore be set properly to ensure that (1) portions of the load on a highly loaded server are migrated to a less loaded server and (2) highly loaded servers do not repeatedly shed load by migrating their load to less loaded servers, thereby incurring unnecessary communication overhead. Since the benchmarking experiments used in this paper cover different workload scenarios, the threshold values used are stated before each experiment in Section 4.5.
- **Migration threshold percentage values**, which can be set for the LoadMinimum load balancing strategy. The *migration threshold* determines the load difference needed between the load at (1) the most heavily loaded location and (2) the least loaded location to trigger the migration of the load from the heavily loaded location to the least loaded location.
- **Dampening value**, which determines what fraction of a newly reported load is considered when making load balancing decisions. Cygnus's *dampening value* can range between 0 and 1 (exclusive). The higher the value, the lower the percentage of newly reported load taken into account by Cygnus when analyzing that load. In particular, the dampening value affects how sensitive Cygnus is to changes in loads. A dampening value of 0.1 was chosen for these experiments so that Cygnus used a higher fraction of the server load at a particular moment. Our goal was to provide Cygnus with finer grained load analysis, while simultaneously reducing its sensitivity to large changes in load.

4.4 Cygnus Overhead Measurements

Figure 2 shows how a client request is intercepted by Cygnus's load balancer component, which then binds the request to a server it choses. Subsequent requests will then be issued directly to the chosen server, as discussed in Sec-

tion 3.3. Below we present empirical results that show the overhead incurred in a load balanced application by this initial call to Cygnus's load balancer. The number of clients in these experiments ranged from 2, 4, 8, and 16, where each client made 500,000 invocations to a single server. The experiments were repeated for `Latency` test configurations (Section 4.3) that included no load balancing support (the baseline), as well as load balancing support using Cygnus's RoundRobin, Random, LeastLoaded, and LoadMinimum strategies.

Figure 6 illustrates how client request throughput varies as the number of clients and servers increase for each configuration described above. This figure shows how request

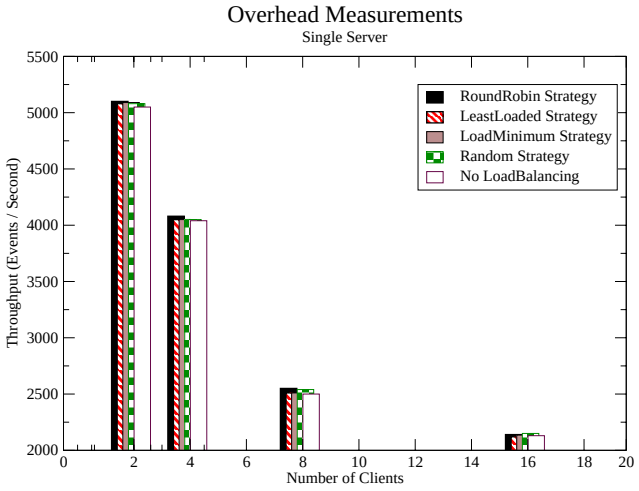

Figure 6. Cygnus Overhead Measurements

throughput decreased for all strategies as the number of clients increased, as expected. Figure 6 also shows, however, that the baseline throughput of the `Latency` test is roughly the same, *i.e.*, the throughput values experienced by clients with and without load balancing support were similar. These results indicate that Cygnus does not add significant overhead to distributed applications.

4.5 Behavior of Different Load Balancing Strategies

Having shown in Section 4.4 that Cygnus does not incur appreciable overhead, we now present results that quantify the behavior of Cygnus's load balancing strategy in different circumstances.

4.5.1 Cygnus Behavior Under CPU-intensive Workloads

Below we evaluate the performance of Cygnus's load balancing strategies in the presence of light and heavy CPU loads. We define a process that incurs light CPU load as a one that does CPU-intensive work but takes comparatively less time to complete than a process that incurs heavy CPU load. Four servers were used in each experiment and the number of clients ranged from 8, 16, and 24 (a larger number of clients than servers was chosen so that very heavy workloads could be imposed on the servers). The clients were divided into two groups: (1) light CPU load generating clients and (2) comparatively heavy CPU load generating client. The experiments were repeated for all Cygnus load balancing strategies, *i.e.*, RoundRobin, Random, LeastLoaded, and LoadMinimum strategies.

We began with an experiment where the clients generated non-uniform loads, specifically light and heavy CPU loads. Figure 7 shows how average client request throughput varied as the number of clients and servers increased for each configuration described above. Load balancing de-

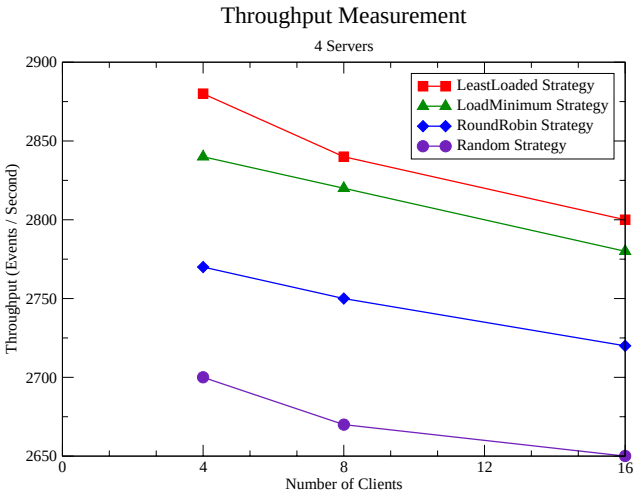

Figure 7. Load Balancing Strategy Performance Under Non-uniform CPU-intensive Loads

cisions in this experiment were based on CPU utilization. The load reporting interval was set to 5 seconds and the reject and critical thresholds for the adaptive strategies were set to 95% and 98% CPU utilization, respectively, which allowed the servers to run at their full potential.

Figure 7 shows the performance difference between the adaptive (LeastLoaded and LoadMinimum) and the non-adaptive (RoundRobin and Random) strategies (only the 4 server cases are shown to avoid cluttering the graph). The goals of this experiment were to (1) compare the performance of the four Cygnus load balancing strategies described in Section 3.2 and (2) demonstrate how the adaptive strategies outperform the non-adaptive ones in the non-uniform load case. In fact, this experiment presents the worst-case behavior for the non-adaptive strategies since two of the four servers always execute heavy CPU load in-

tensive requests, while the other two servers execute the light CPU load intensive requests. Hence, two servers are extremely loaded, whereas the other two servers are less loaded. In contrast, the adaptive strategies utilized all the server resources, thereby improving the average client throughput.

Although our LBPerf testbed can only approximate actual workloads, our results indicate a significant difference in performance between the two types of load balancing strategies. Moreover, the performance advantage of adaptive strategies over non-adaptive ones would be even greater for CPU loads that were more intensive and non-uniform. To illustrate this point empirically, Figure 8 shows the maximum CPU utilization reached for each server in the experiment described above. This figure depicts how non-adaptive

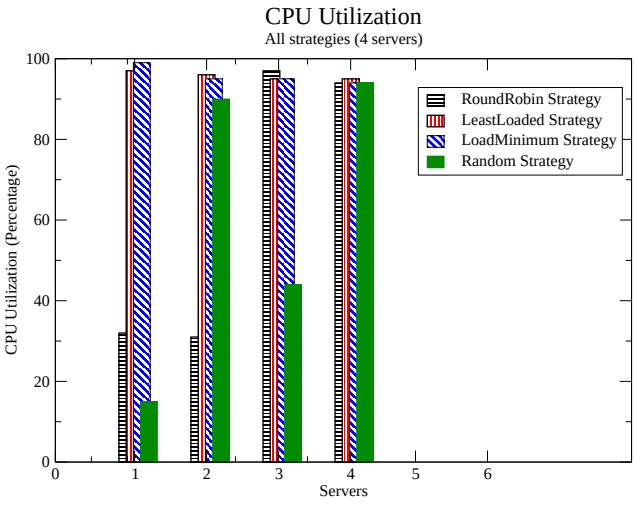

Figure 8. CPU Utilization in Each Server

load balancing strategies can underutilize servers since they do not account for server load conditions. In contrast, adaptive strategies use this system state information to make effective dynamic load balancing decisions that can improve overall performance.

Figure 8 also shows the difference between the performance of the LoadMinimum and LeastLoaded adaptive strategies, which stems from the run-time threshold parameters available to each strategy. For example, as client requests are routed to a location, the LeastLoaded strategy uses the reject threshold value to decide whether to process incoming client requests or reject them. In contrast, the LoadMinimum strategy has no such threshold. Thus, even though both strategies choose the least loaded location to handle client requests, the LeastLoaded strategy has the luxury of using its reject threshold to prevent the system from being overly loaded sooner rather than later.

During periodic load reports, the LeastLoaded strategy determines if the location load has exceeded the critical threshold value. If so, this strategy migrates the particular location's session to the least loaded location, *i.e.*, it forces load shedding to occur. In the LoadMinimum strategy case, conversely, if the least loaded location's load is less than the monitored location's load by a certain percentage the load at the monitored location is migrated to the least loaded location.

In systems with highly non-uniform loads, some servers are loaded for short periods of time, whereas others are loaded for long periods of time. The LoadMinimum strategy will therefore try repeatedly to shed load from heavily loaded location's to less loaded locations. Although the number of migrations could be controlled by using a higher percentage value, some migrations are inevitable. The performance of LoadMinimum strategy in the presence of non-uniform loads will therefore generally be less than the performance of the LeastLoaded strategy, as shown in Figure 8.

We do not consider uniform loads in the above discussion since non-adaptive strategies can trivially perform as well as adaptive strategies. Our results and analysis show that the performance of the LeastLoaded strategy is better than the LoadMinimum strategy in the presence of non-uniform loads. Moreover, our results show how a thorough understanding of the run-time parameters associated with the LeastLoaded and LoadMinimum strategies is needed to set them appropriately. In the remainder of this section, we use LBPerf to study the importance of these parameters empirically.

4.5.2 Importance of the Migration Threshold Value

As discussed in Section 4.3.1, the performance of the LoadMinimum strategy depends heavily on the threshold value chosen. The chosen threshold value affects the number of migrations made to complete a particular client session. It is worthwhile, however, to consider whether moving a session to another location will speedup the completion of that session.

According to [23], session migrations are not always as effective as the initial session placement decisions, which suggests that careful analysis of session migration implications should be made before migrating a client session. Such observations are hard to validate, however, without proper empirical study of the scenario in an experimental testbed. We therefore used LBPerf to design an experiment that switched the threshold values and measured the number of migrations and the resulting throughput. We repeated this experiment for two different cases: (1) when the threshold value was 40% and (2) when the threshold value was 80%. We chose 80% to prevent or minimize client session migrations and 40% to determine what happens when a client session at a fully loaded location is migrated to a location with half the load.

To illustrate the performance implications of choosing different migration thresholds, Figure 9 shows the average client throughput achieved using the LoadMinimum strategy by changing the threshold values as the number of clients were varied between 4, 8, 16, and 32 (again, only results for 4 the server case are shown to avoid graph clutter). This figure shows the difference in the performance is

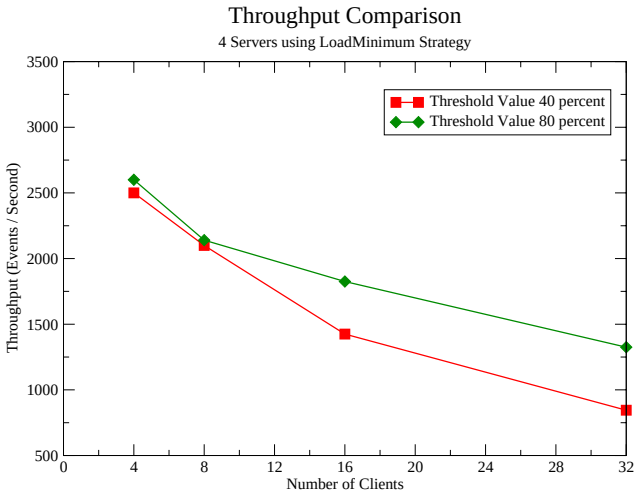

Figure 9. Performance of the LoadMinimum Strategy Under Different Threshold Values

insignificant when the load is very light, *i.e.*, when the difference between the number of clients and the number of servers is small. As the number of clients grows more than the number of servers, however, the increased load causes more migrations in the 40% case. Hence, the throughput difference with the 80% case is more significant.

These results should not be interpreted as implying that client session migrations are always bad, however, since in certain cases client session migrations help to shed load at heavily loaded locations to less loaded ones. An example is a scenario with non-uniform system loads, where the difference between the generated loads is so large that one type of workload takes a long time to complete, whereas another type of workload takes much less time to complete. In such cases, migrating the load from the heavily loaded location to the less loaded location can improve response time. The threshold value should therefore be selected based on (1) the request completion time, (2) type of servers in the system (which may be homogeneous or heterogeneous), and (3) the existing load in the system. Our future work will devise self-adaptive load balancing strategies that can tailor threshold values according to run-time state information in the system.

4.5.3 Importance of the Reject and Critical Threshold Values

Figure 7 shows the performance of the LeastLoaded and the LoadMinimum strategies in the presence of non-uniform CPU intensive loads. The configurations for that experiment were set in such a way that the servers performed at 100% CPU utilization. There are certain cases, however, where rather than overloading the system, we need to *bound* CPU utilization so that certain QoS properties (such as latency and response time) are met predictably.

As discussed in Section 4.3.1, the LeastLoaded strategy has two threshold values, *reject* and *critical*, that determine whether a server can be considered to receive request and when to migrate requests from one server to another, respectively. These thresholds help ensure control of the system utilization by the strategy. Although these thresholds may cause some requests to be rejected, such admission control may be needed to meet end-to-end QoS requirements.

We used LBPerf to design an experiment that switched the reject threshold values and measured the number of missed requests and the resulting throughput. The load reporting interval was set at 5 seconds and the reject and critical thresholds for the adaptive strategies were set at 75% and 98% CPU utilization, respectively. We chose these values to test what happens when the load at each server is kept at an optimal level (75% for this case) versus being very high (98%).

To illustrate the performance implications of choosing different reject threshold values, Figure 10 shows the average client throughput achieved using the LeastLoaded strategy by changing the threshold values as the number of clients were varied between 4, 8, 16 and 32 (again, only results for the four server cases are shown to avoid graph clutter). This figure shows the difference in the performance is insignificant when the load is very light, *i.e.*, when the difference between the number of clients and servers is small. As the number of clients grows in our experiment, however, the increased load causes more request rejections in the 75% case. Hence, the throughput difference with the 98% case is more significant.

4.6 Summary of Empirical Results

The empirical results in this section use LBPerf to help quantify how Cygnus's adaptive load balancing strategies can improve the distributed system scalability without incurring significant run-time overhead. Our results suggest that in the presence of non-uniform loads, the adaptive load balancing strategies outperform the non-adaptive load balancing strategies. By empirically evaluating Cygnus's adaptive load balancing strategies, we show that the LeastLoaded strategy performs better than the LoadMinimum strategy under such load conditions, which motivates the

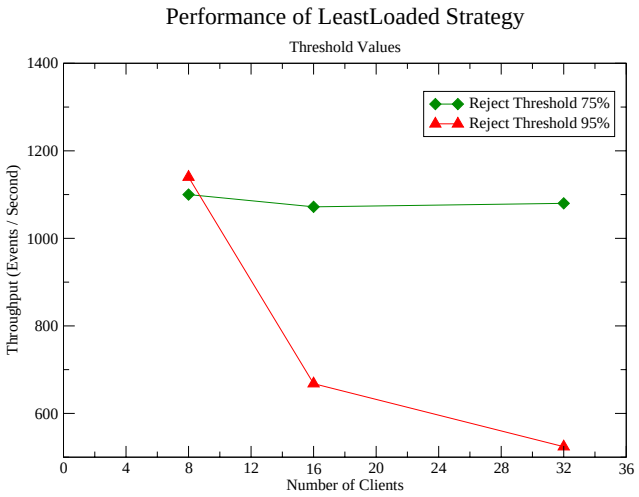

Figure 10. Performance of the LeastLoaded Strategy Under Different Threshold Values

choice of the LeastLoaded strategy in those conditions. We also showed the importance of (1) reducing the number of client request migrations and (2) maintaining system utilization to enable more predictable behavior from the adaptive load balancing strategies.

5 Related Work

Load balancing middleware can enhance the flexibility of many types of distributed applications [7]. For example, load balancing middleware can take distributed system behavior and state into account when making load sharing and distribution decisions, enable flexible application-defined selection of load metrics, and can also make load balancing decisions based on request content.

Some load balancing middleware implementations [15, 10] integrate their functionality into the Object Request Broker (ORB) layer of the middleware itself, whereas others [9, 2, 1] implement load balancing support at a higher level, such as the common middleware services layer. The remainder of this section compares and contrasts our research on Cygnus with related efforts on CORBA load balancing middleware that are implemented at the following layers in the OMG middleware reference architecture.

- **Service layer.** Load balancing can 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 via a hierarchy of *event channels* that fan out from event source *suppliers* to the event sink *consumers*. [2] extends the CORBA Naming Service to add load balancing capabilities. When a client calls the `CosNaming::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. 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.

  Various commercial CORBA implementations provide service-based 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 [3] uses a per-request load balancing approach, also described in [18]. In contrast, Cygnus does not incur the per-request network overhead of the BEA approach, yet can still adapt to dynamic changes in the load.

- **ORB layer.** Load balancing can also be implemented within the ORB itself, rather than as a service. Borland's VisiBroker implements an ORB-based load balancing strategy, where Visibroker's object adapter [10] 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-based load balancing mechanisms is that the amount of indirection involved when balancing loads can be reduced due to the close coupling with the ORB *i.e.*, the length of communication paths is shortened. The disadvantage of ORB-based load balancing, however, is that it requires modifications to the ORB itself. Until such modifications are adopted by the OMG, they will be proprietary, which reduces their portability and interoperability. The Cygnus service-based load balancer therefore does not rely on ORB extensions or non-standard features, *i.e.*, it does not require any modifications to TAO's ORB core or object adapter. Instead, it implements adaptive load balancing via standard CORBA mechanisms, such as *servant locators*, *server request interceptors*, and *IOR interceptors*. Unlike ORB-based load balancing approaches, however, Cygnus only uses standard CORBA features, so it can be ported to any C++ CORBA ORB that implements the CORBA specification.

6 Concluding Remarks

Load balancing middleware is an important technology for improving the scalability of distributed applications. This paper describes Cygnus, which is a middleware load balancing and monitoring service designed to satisfy the requirements specified in Section 2. Cygnus provides a framework for integrating a range of adaptive and non-adaptive load balancing strategies, such as *RoundRobin*, *Random*, *LeastLoaded*, and *LoadMinimum*, to help increase overall system scalability for many types of CORBA middleware-based distributed applications in an efficient, transparent, and portable manner.

Our empirical results in this paper use the LBPerf benchmarking toolkit to show how various load balancing strategies supported by Cygnus allow distributed applications to be load balanced efficiently. Cygnus increases the scalability of distributed applications by balancing requests across multiple back-end server members without increasing round-trip latency substantially. The adaptive load balancing strategies implemented in Cygnus generally handle non-uniform workloads much better than its non-adaptive strategies. The threshold values configured with Cygnus's adaptive load balancing strategies help maintain various QoS properties, such as throughput, latency, and CPU utilization.

Our empirical benchmarks also revealed how tuning threshold parameters is essential to improve the performance and scalability of distributed applications. We therefore plan to focus our future work on *self-adaptive* load balancing strategies, which dynamically tune these threshold values according to the run-time state of distributed applications they manage.

References

[1] M. Aleksy, A. Korthaus, and M. Schader. Design and Implementation of a Flexible Load Balancing Service for CORBA-based Applications. In *Proc. of the International Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*, Las Vegas, USA, June 2001.

[2] T. Barth, G. Flender, B. Freisleben, and F. Thilo. Load Distribution in a CORBA Environment. In *Proc. of the International Symp. on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, Sept. 1999. OMG.

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

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

[5] S. P. Dandamudi and M. K. C. Lo. A Comparitive study of Adaptive and Hierarchical Load sharing policies for Distributed Systems. In *International Conference on Computers and Their Applications*, Mar. 1998.

[6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Tran. on Software Engineering*, 12(5):662–675, May 1986.

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

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

[9] K. S. Ho and H. V. Leong. An Extended CORBA Event Service with Support for Load Balancing and Fault-Tolerance. In *Proc. of the International Symp. on Distributed Objects and Applications (DOA'00)*, Antwerp, Belgium, Sept. 2000. OMG.

[10] I. 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. `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] W. G. Krebs. Queue Load Balancing / Distributed Batch Processing and Local RSH Replacement System. `www.gnuqueue.org/home.html`, 1998.

[14] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[15] M. Lindermeier. Load Management for Distributed Object-Oriented Environments. In *Proc. of the* 2^{nd} *International Symp. on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, Sept. 2000. OMG.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, Dec. 2002.

[17] O. Othman, J. Balasubramanian, and D. 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.

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

[19] O. Othman and D. 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.

[20] D. C. Schmidt, R. Schantz, M. Masters, J. Cross, D. Sharp, and L. DiPalma. Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. *CrossTalk*, Nov. 2001.

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

[22] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations.

IEEE Transactions on Software Engineering, 16(3):360–369, Mar. 1990.

[23] W. Zhu, C. F. Steketee, and B. Muilwijk. Load Balancing and Workstation Autonomy on Amoeba. *Aust. Computer Science Communications*, 17(1):588–597, Feb. 1995.