The Design of an Adaptive Middleware Load Balancing and Monitoring Service

Ossama Othman[1], Jaiganesh Balasubramanian[1], and Douglas C. Schmidt[2]

[1] {ossama,jai}@doc.ece.uci.edu
Dept. of Electrical and Computer Engineering
University of California
608 Engineering Tower
Irvine, CA 92697, USA

[2] d.schmidt@vanderbilt.edu
Institute for Software and 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, and (2) it describes the capabilities and design of Cygnus, which is an adaptive load balancing service. The findings in this paper show that adaptive middleware load balancing is a viable solution for improving the scalability of distributed applications.

1 Introduction

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

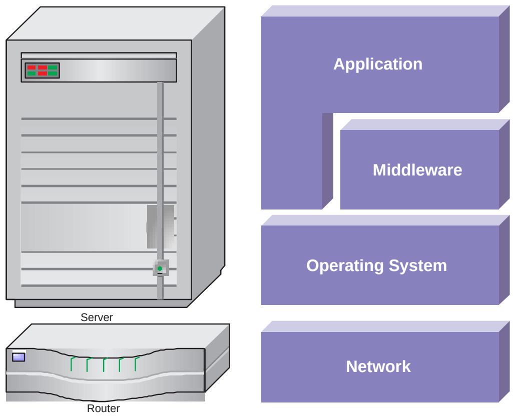


Fig. 1. Load Balancing Layers

load balancing is often provided by routers and domain name servers [1]. OS-level load balancing is generally provided by clustering software [2]. Application-level load balancing is performed by the application itself [3]. 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.

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* [4], such as CORBA [5] or Java RMI [6]. 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 [7, 8]. 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 [9, 10], including our own earlier work [11, 12] 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 [13] 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 specify how to tackle load balancing capabilities required by distributed systems architects and developers. Cygnus is available with *The ACE ORB* (TAO) [14] version 5.3, which implements the CORBA 3.0 specification [5]. 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; and Section 4 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 [13]. Sidebar 1 defines and illustrates the load balancing concepts and components[3] 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) Cygnus and the objects to be load balanced (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.

2.2 Resolving Load Balancing Challenges with Cygnus

Figure 2 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*

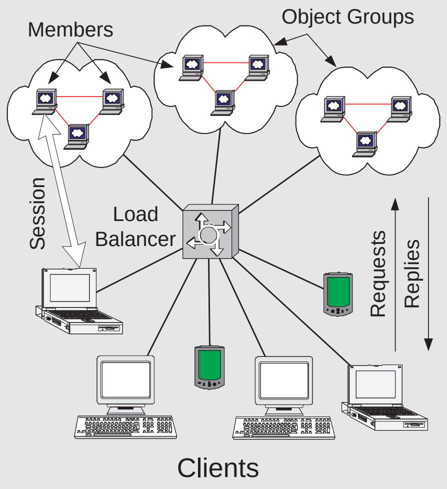
[3] In this paper, the term *component* is used generically, *i.e.*, an identifiable entity in a program, rather than more specifically, *e.g.*, a component in the CORBA Component Model [15].

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:



manager, 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 2 outline the elements of the Cygnus, they do not motivate what these elements do or more importantly *why* they are impor-

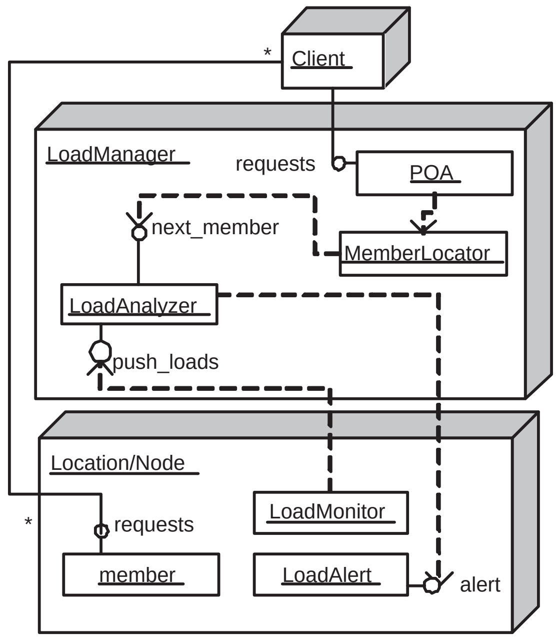


Fig. 2. Components in the Cygnus LB/M Service

tant. 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.[4] 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 [8], 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.

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

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 [12].

Solution $\rightarrow$ 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 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 [16] 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) [17] are used to minimize the amount of time other operations are blocked waiting for the adaptive load balancing operations to complete.

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 [11]), 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 $\rightarrow$ *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 3, a load monitor can be configured with either of the following two policies:

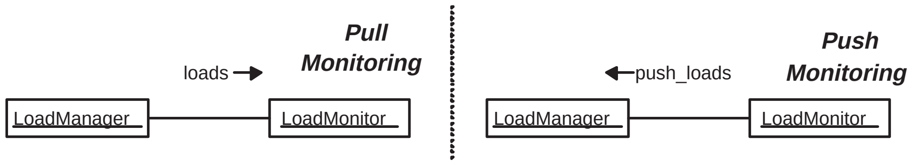


Fig. 3. Load Reporting Policies

- *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 [12].

Solution $\rightarrow$ 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[5] 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

[5] A Portable Interceptor is an instance of the Interceptor design pattern [18], 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.

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.[6]

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 [11], whose interactions are shown in Figure 4. Each interaction in Figure 4 is outlined below.

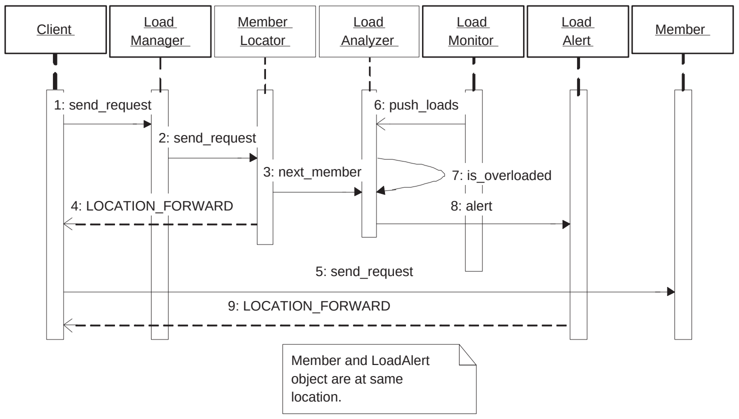


Fig. 4. Cygnus Load Balancing and Monitoring Interactions

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 [11]) 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.

[6] 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.

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* 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 The Design of Cygnus: the TAO CORBA Load Balancing Service

This section describes the design of the Cygnus adaptive load balancing service that is distributed with The ACE ORB (TAO) [14] (which is a CORBA-compliant ORB that supports applications with stringent QoS requirements). TAO’s Cygnus load balancing service makes it easier to develop distributed applications in heterogeneous environments by providing application transparency, high flexibility, scalability, run-time adaptability, and interoperability. The Cygnus load balancing service drove the model, architecture, and content of the proposed CORBA LB/M specification.

3.1 Design Challenges and Solutions

The following design challenges were identified during the development of *Cygnus*:

1. Enhancing load control
2. Supporting modular load balancing strategies
3. Complete server transparency and
4. Maximizing throughput and minimizing network and resource overhead.

The solutions that were applied to address each of these challenges are discussed below. These solutions manifest themselves within the load balancing service components described in Section 2.2.

.

Challenge 1: Enhancing Load Control

Context. A load balancing service performs load control decisions and actions based on load feedback.

Problem. A load balancer must react to various load conditions to ensure that loads across members are balanced. For example, when high load conditions occur, a member must be instructed to forward the client request back to the load balancer so that subsequent requests can be reassigned to a less loaded member. However, this should be done in a manner that transparent to both clients and servers.

Solution → the Mediator pattern. A load alert component responds to load balancing requests sent by the load balancer. Depending on the type of request the load balancer sends to the load alert component, the member will either be forced to continue accepting client requests or redirect the client back to the load balancer. The load balancer never interacts with the member directly – all interaction occurs via the load alert component, as shown in Figure 5. Similarly, the member never interacts with the load balancer directly. The `LoadAlert` component *mediates* all load control interactions with the load balancer.

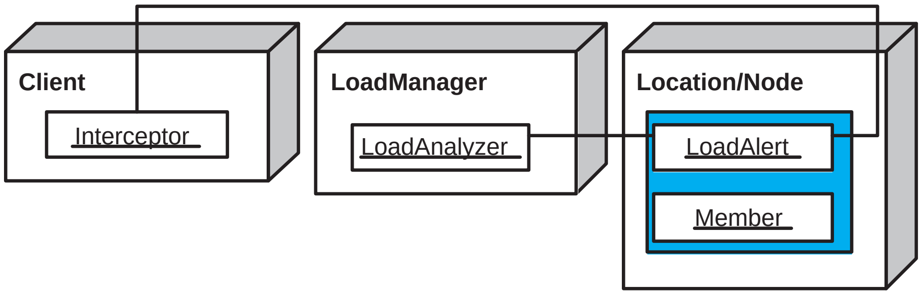


Fig. 5. The `LoadAlert` Mediator

Applying the solution in Cygnus. When enabling adaptive load balancing in a particular distributed application, a load monitor (in the "pull" monitoring case) and a load component are registered with the load balancer. As shown in Figure 6, the load bal-

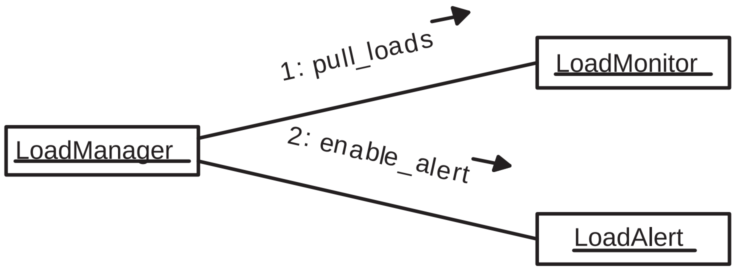


Fig. 6. Feedback and Control When Balancing Loads

ancer queries the load monitor for the load at the location the current member resides at, assuming that pull-based load monitoring is being used (see Section 2.2). In other words, the load balancer receives *feedback* from the load monitor. Load balancing control requests–called *load alerts*–are then sent to the load alert component from the load balancer to set the “alert” state of the member’s location to one of the following values when load shedding, *i.e.*, reduction in load, is either unnecessary or necessary:

– *Not Alerted* – When load shedding is *not* required, the member continues to accept requests.
– *Alerted* – When load shedding is required to reduce the load at the location, an “alert” causes the load alert component to redirect client requests back to the load balancer, at which point the load balancer forwards the request to a less loaded member.

The load shedding interactions are depicted in Figure 7. This figure shows the two

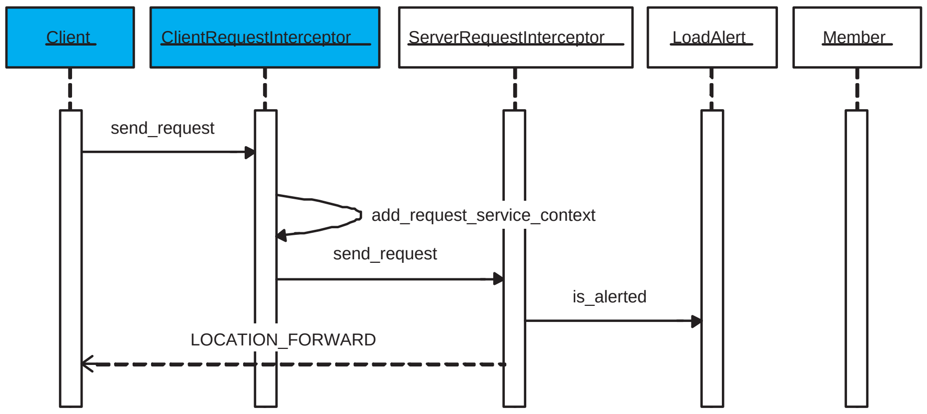


Fig. 7. Load Shedding Interactions

additional entities that were not discussed previously:

– the `ClientRequestInterceptor` and
– the `ServerRequestInterceptor`.

These entities expose a standard CORBA interface and are used in the figure to illustrate how to transparently and portably shed load. Descriptions of how they are used in Cygnus’ overall load shedding interactions follow:

1. A client request is intercepted by the `ClientRequestInterceptor`.
2. The `ClientRequestInterceptor` determines that the target is a load balanced one based on pre-configured application settings. It injects the “out-of-band” data that identifies the target object as a load balanced one.
3. The client request is allowed to proceed.

4. The `ServerRequestInterceptor` checks the “out-of-band” data for the identification information injected on the client side, and if the load alert component has been told that it should reject requests.
5. If the information exists and requests are to be rejected, the appropriate exception will be issued by the `ServerRequestInterceptor` to force the client to re-invoke its request on the load balancer.

Armed with a load monitor and load alert component, such a load balancer is *adaptive* due to the bidirectional feedback/control channel between the load monitor, load alert component and the load balancer. Since the load monitor is decoupled from the load balancer it is also possible to balance loads across locations, and hence members, based on various types of load metrics. For instance, one type of load monitor could report CPU loads, whereas another could report I/O resource load or both. The fact that the type of load presented to the load balancer is opaque allows the same load balancer–specifically the load analysis algorithm–to be reused for any load metric.

Challenge 2: Supporting Modular Load Balancing Strategies

Context. A distributed system employs a load balancing service to improve overall throughput by ensuring that loads across locations are as uniform as possible. In some applications, loads may peak in a predictable fashion, such as at certain times of the day or days of the week. In other applications, loads cannot be predicted easily *a priori*.

Problem. Since certain load analysis techniques are not suitable for all use-cases, it may be useful to analyze a set of location loads in different ways depending on the situation. For example, to predict future location loads it may be useful to analyze the history of loads at locations where members of given object group reside, thereby anticipating high load conditions. Conversely, this level of analysis may be too costly in other use-cases, *e.g.*, if the duration of the analysis exceeds the time required to complete client request processing.

In some applications it may even be necessary to change the load analysis algorithm dynamically, *e.g.*, to adapt to new application workloads. Moreover, bringing the system down to reconfigure the load balancing strategy may be unacceptable for applications with stringent 24$\times$7 availability requirements. Likewise, application developers may be interested in evaluating several alternative load balancing policies, in which case requiring a full recompilation or relink cycle would unduly increase system development effort. A load balancing service cannot simply implement all possible load balancing strategies, however, *e.g.*, application developers may wish to define application-specific or *ad-hoc* load balancing algorithms during testing or deployment.

So, how can we allow dynamic (re)configurations of the load balancing service, such as the load monitor and load analyzer, without requiring expensive system recompilations or interruptions of service?

Solution $\rightarrow$ the Strategy pattern. The *Strategy* design pattern, as mentioned earlier, allows applications to install and uninstall different behavior run-time. In the proposed CORBA LB/M service this pattern can be used to change the member selection strategy dynamically. A load balancer can therefore use this pattern to adapt to different load balancing use-cases, without being hard-coded to handle those specifically.

At times it may be necessary to load balance only a few members, in which case a simple load balancing strategy may suffice. In other situations, such as during periods of peak activity during the workday, a load balancing strategy may need modifications to account for increased load. In such cases, a more complex strategy may be necessary. The Strategy pattern makes it easy to dynamically configure load balancing algorithms appropriate for different use-cases *without* stopping and restarting the load balancer.

Applying the solution in Cygnus. The load analyzer uses the Strategy pattern to customize the load balancing algorithm used when making load balancing decisions, as depicted in Figure 8. The proposed OMG load balancing service can be configured

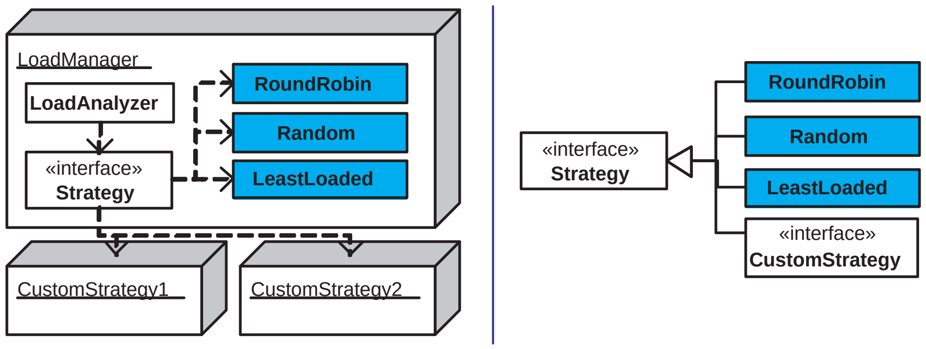


Fig. 8. Applying the Strategy Pattern to the OMG Load Balancing Service

dynamically to use the following *built-in* strategies:

- **Round-robin** – This non-adaptive strategy is straightforward and does not take load into account. Instead, it simply causes a request to be forwarded to the next member in the object group being load balanced [8].
- **Random** – This non-adaptive strategy also does not take load into account. It simply forwards clients requests to an object group member residing at a random location. Of course, only locations with members residing at them are considered for selection.
- **Least loaded** – This adaptive strategy is more sophisticated than the round-robin and random algorithms described above. The goal of this strategy is to ensure load differences fall within a certain tolerance, *i.e.*, it attempts to ensure that the average difference in load between each location/member is minimized. The member at the least loaded location is selected.[7]

[7] An earlier, less refined, version of this load balancing strategy first appeared in TAO's initial load balancer prototype. That balancing strategy was called *Minimum Dispersion.*

The proposed CORBA LB/M specification is not limited to these built-in strategies, however. Any custom strategy unknown to the load balancer may be "plugged in" at any point during the load balancer's lifetime since all strategies, including the built-in ones, implement the same strategy interface. A large amount of work on load balancing strategies [19] has already been done. Many of those same strategies can be integrated in to the load balancing service via the Strategy pattern implementation described above.

Challenge 3: Complete Server Transparency

Context. Distributed applications can suffer from poor performance due to a bottleneck at a single overloaded server. To address this performance bottleneck, an *adaptive* load balancing service is used to (1) distribute client requests equitably among a group of members and (2) actively monitor and control loads on members in that group.

Problem. An adaptive load balancing service must communicate with members so it can force them to either accept or reject requests. To achieve this level of communication, application servers must be programmed to accept load balancing requests (as well as client requests) from the adaptive load balancing service. However, most distributed applications are not designed with this ability, nor should they necessarily be designed with that ability in mind since it complicates the responsibilities of application developers.

Solution $\rightarrow$ the Component Configurator and Interceptor Patterns. If adaptive load balancing is to be used transparently on the server-side of a distributed application, there must be some way to install feedback/control mechanisms into the server without altering the server application software. Fortunately, most ORB middleware–and in particular CORBA–provide a meta-programming mechanism based on the Interceptor pattern [18]. These mechanisms can alter the behavior of a client or a server when processing a given client request [20]. An interceptor can be installed at run-time to provide the functionality necessary to (1) communicate with the load balancing service and (2) accept load control requests from the load balancing service. Since the interceptor mechanism is part of the middleware implementation, server application software need not be modified.

To provide true server-side transparency, however, there must be some means of installing interceptors transparently to control requests from the adaptive load balancing service. The Component Configurator pattern [18] can be used to dynamically load a service into an application at run-time. In particular, a Component Configurator can be used to transparently install a load balancing interceptor into an application's underlying middleware at run-time, as illustrated in Figure 9. Using this approach, the overall throughput of a distributed application can be improved without modifications to distributed application server code.

Applying the Solution in Cygnus. The functionality required to install a load balancing interceptor transparently at run-time is available in most CORBA ORBs, such as TAO. This functionality includes *portable interceptors* and the *CORBA Component Model*, as outlined below:

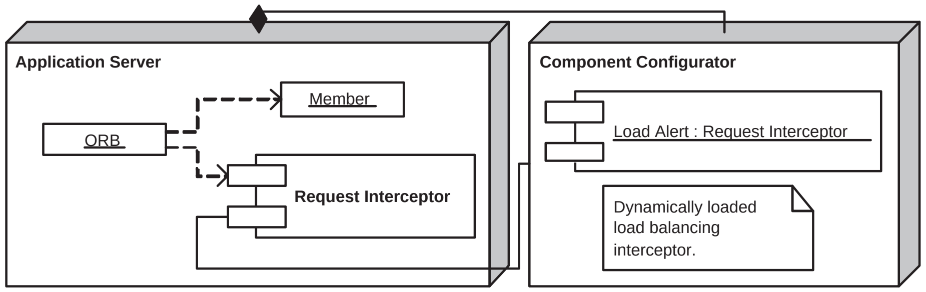

Application Server
Member
ORB
Request Interceptor
Component Configurator
Load Alert : Request Interceptor
Dynamically loaded load balancing interceptor.

Fig. 9. Transparent Server-side Adaptive Load Balancing

- Portable interceptors: Portable interceptors [5] can capture client requests transparently before they are dispatched to an object group member. For example, a *server request interceptor* could be added to the ORB where a given member runs. Since interceptors reside within the ORB no modification to server application code is necessary, other than registering the interceptor with the ORB when it starts running.
- CORBA Component Model (CCM): The CCM [21] introduces *containers* to decouple application component logic from the configuration, initialization, and administration of servers. In the CCM, a container creates the POA[8] and interceptors required to activate and control a component. These are the same CORBA mechanisms used to implement the server components in TAO's load balancing service. The standard CCM containers can be extended to implement automatic load balancing *generically* without changing application component behavior.

Challenge 4: Maximizing Throughput and Minimizing Network and Resource Overhead

Context. A distributed application is suffering from degraded performance due to limited resources. This is basically the same scenario used in Section 3.1.

Problem. Simply integrating a load balancing service into a distributed application does not necessarily mean that performance will improve significantly. This is particularly true if the load balancing service implementation has its own inefficiencies. For instance, it may continuously attempt to make load balancing decisions despite the fact that no additional client invocations have been made to perturb the overall load conditions. Such an implementation would typically be slower in making load balancing decisions under its own heavy load. Moreover, the increased load analysis more than likely requires the load balancer to query loads at all locations it is aware of. This increases network utilization, for example, more than necessary and leaves less bandwidth available for the application being load balanced.

[8] The Portable Object Adapter (POA) is responsible for dispatching client requests to the intended target server.

Solution → lazy evaluation and asynchronous method invocation. The *lazy evaluation* approach can be used to reduce the self-incurred load caused by the load balancer's load analysis. Specifically, load analysis will only occur when it is necessary to bind a client to an object group member. Basically, a client invocation on an object group through the load balancer will trigger the load analysis and shedding process to occur.

However, this lazy evaluation approach has the disadvantage where the client must wait for the complete load analysis and shedding procedure to complete before it can be forwarded to the actual member. The load analysis wait cannot be avoided since it is an integral part of the member selection process. The load shedding procedure, on the other hand, can be performed in parallel. It need delay the client from being forward to the actual member. Load shedding can be an expensive procedure since it requires that the load balancer make invocations on the typically remote `LoadAlert` component described in Section 3.1.

One technique to avoid this delay is to use non-blocking CORBA *one way* invocations. Such invocations are not guaranteed to arrive at the intended target, however, nor is it possible to convey exceptional conditions back to the load balancer. The ability to determine the health of the remote `LoadAlert` component is important since load shedding is not possible without it.

A better way to avoid delaying the client forward is to use CORBA standard *asynchronous method invocations* (AMI) [17]. AMI allows an invocation to be made asynchronously without blocking the caller, such as the client in the above scenario, until a reply from the invocation target arrives. Not only does it avoid the delays, it also allows exceptional conditions to be reported back to the load balancer.

Using both the lazy evaluation and AMI approach allows load balancing decisions (member selection) and load balancing control (load shedding) procedures to be completed in parallel, which reduces resource utilization and improves the ability of the load balancer to bind clients to members more quickly.

Yet another approach would be to spawn a separate thread to handle load shedding in parallel. Doing so, however, may be costly in terms of thread activation overhead. Certainly, pre-activation of the thread will help but not all platforms support threads. In those cases, AMI is currently the only portable solution.

Applying the Solution in Cygnus. Applying lazy evaluation to TAO's load balancer is relatively straightforward. Load analysis, member selection and load shedding functions are simply not called until a client makes an invocation on the load balancer. After the member locator is invoked, load analysis, member selection and load shedding begin.

Incorporating AMI into the remote load shedding invocations is also straightforward. A reply handler is implemented to handle the asynchronously returned replies, and the synchronous load shedding method calls were replaces by their asynchronous counterparts; the only difference being an additional reply handler callback[9] parameter passed to them.

[9] AMI requires that a reply handler be supplied so that it may be called on when the invocation reply returns.

4 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 run-time 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 [14].

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 specification that interoperates with the Cygnus C++ implementation provided with TAO.

References

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

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

[3] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," in *Proceedings of the USENIX Summer Conference*, USENIX, June 2000.

[4] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2002.

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

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

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

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

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

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

[12] O. Othman, C. O'Ryan, and D. C. Schmidt, "Designing an Adaptive CORBA Load Balancing Service Using TAO," *IEEE Distributed Systems Online*, vol. 2, Apr. 2001.

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

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

[15] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

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

[17] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

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

[19] C.-C. Hui and S. T. Chanson, "Improved Strategies for Dynamic Load Balancing," *IEEE Concurrency*, vol. 7, July 1999.

[20] N. Wang, K. Parameswaran, and D. C. Schmidt, "The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware," in *Proceedings of the 6th Conference on Object-Oriented Technologies and Systems*, (San Antonio, TX), pp. 103–118, USENIX, Jan/Feb 2000.

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