taolbCygnus

The Design, Optimization, and Performance of an Adaptive Middlware Load Balancing Service

by

Ossama Othman

B.Eng. (City College of New York) 1996

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Electrical and Computer Engineering

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, IRVINE

Committee in charge:
Professor Douglas C. Schmidt, Chair
Professor Stephen Jenks
Professor Raymond Klefstad

Fall 2002

The dissertation of Ossama Othman is approved:

Chair Date

Date

Date

University of California, Irvine

Fall 2002

The Design, Optimization, and Performance of an Adaptive Middlware Load Balancing Service

Abstract

The Design, Optimization, and Performance of
an Adaptive Middlware Load Balancing Service

by

Ossama Othman
Master of Science in Electrical and Computer Engineering

University of California, Irvine

Professor Douglas C. Schmidt, Chair

Distributed object computing (DOC) 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 balance system processing load among multiple servers. Load balancing can help improve overall system scalability by ensuring that client application requests are distributed and processed equitably across groups of servers.

Earlier generations of load balancing middleware services were simplistic since they only addressed specific use-cases and environments. These limitations made it hard to use the same load balancing service for anything other than a small class of distributed applications. This lack of generality forced continuous redevelopment of application-specific load balancing services. Not only did this redevelopment increase distributed applications deployment costs, but it also increased the potential of producing non-optimal load balancing implementations since time-proven load balancing service optimizations could not be reused directly without undue effort.

This thesis presents the following contributions to research on load balancing techniques for DOC middleware:

1. It describes deficiencies with common load-balancing techniques, such as introducing unnecessary overhead or not adapting dynamically to changing load conditions.
2. It presents a novel adaptive load balancing service called that can be implemented efficiently using the capabilities of CORBA, which is a widely used, standards-based DOC middleware specification.
3. It explains how alleviates existing middleware load balancing services limitations, such as lack of server-side transparency, centralized load balancing, sole support for stateless replication, fixed load monitoring granularities, lack of fault tolerant load balancing, non-extensible load balancing algorithms, and simplistic replica management.
4. It discusses the forthcoming OMG Load Balancing and Monitoring specification, which is based on the research conducted for this thesis.
5. It describes the key design challenges faced when integrating the load balancing service in the The ACE ORB (TAO) and how these challenges were resolved by applying patterns.
6. It presents the results of benchmark experiments that empirically evaluate different load balancing strategies by measuring the overhead of each strategy and showing how well each strategy balances system load.

Professor Douglas C. Schmidt
Dissertation Committee Chair

To my ...,

INSERT NAME,

******** SAY SOMETHING NICE :-) **********

Contents

List of Figures

List of Tables

Acknowledgments

Thank someone!

Part I

Thesis

Chapter 1

Introduction

Scalability is crucial for many distributed applications. Load balancing is a promising techique for improving the scalability of distributed applications. Simply introducing load balancing into a distributed application, however, may not necessarily fulfill its scalability requirements due to inadequacies in the load balancing architecture. This chapter (1) motivates the need for load balancing and introduces key concepts used throughout this thesis, (2) outlines problems with existing load balancing architectures, (3) enumerates the key research challenges, solutions, and contributions addressed by this thesis, and (4) describes how these solutions can be applied to production distributed applications.

1.1 Motivation

As the demands of resource-intensive distributed applications have grown, the need for improved overall scalability has also grown. This section presents an example of such an application, in addition to a candidate solution that can fulfill the needs of that example.

1.1.1 A Distributed Stock Trading Example

Consider the online stock trading system shown in Figure **??**.

A distributed online stock trading system creates sessions through which trading is conducted. This system consists of multiple back-end servers that process session creation requests sent by clients over a network. Each back-end server performs the same tasks.

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

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

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

- Increase the scalability of the system;
- Reduce the initial investment when the number of clients is small; and

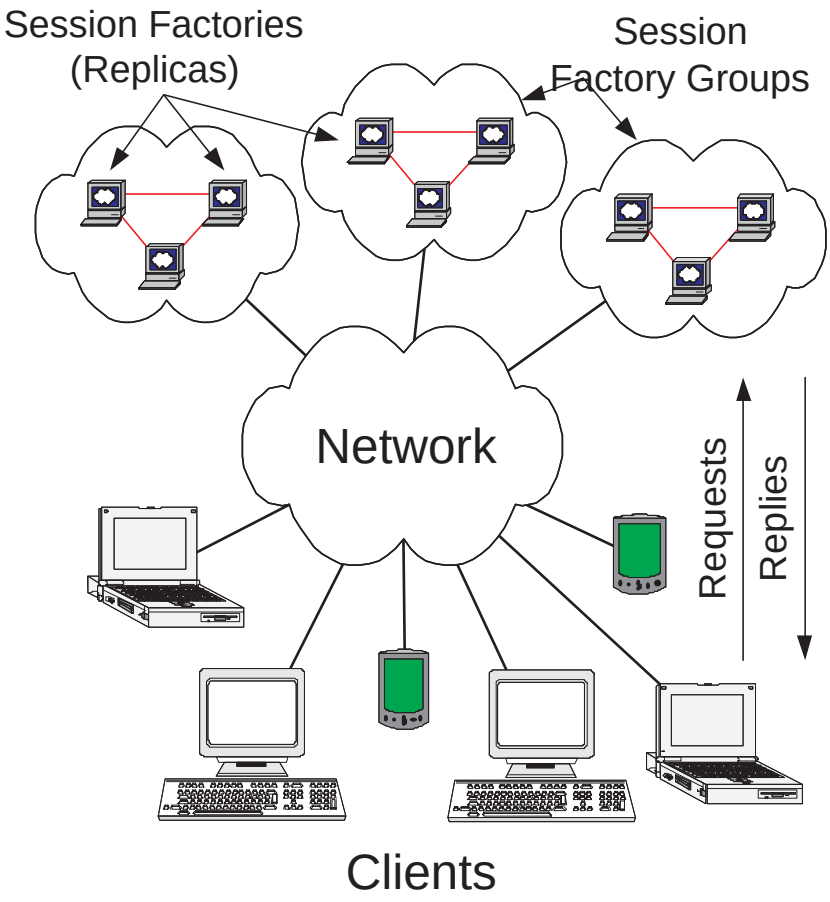

Figure 1.1: A Distributed Online Stock Trading System

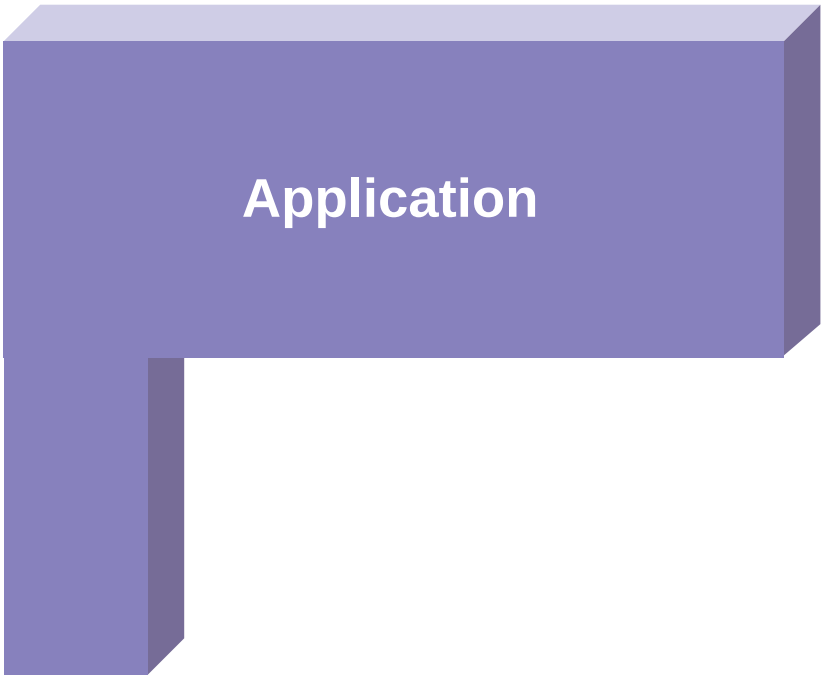

Figure 1.2: Load Balancing Layers

- Allow the system to scale up gracefully to handle more clients and processing workload in larger configurations.

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

1.1.2 Candidate Solutions

As described in the example in Section **??**, load balancing is useful for distributed applications with high scalability requirements. Load balancing is typically performed in the following platform levels or layers:

- Network
- Operating System
- Middleware
- Application

These are depicted in Figure **??**.

Network level load balancing is often provided by routers and name servers. Operating system level load balancing is generally provided by clustering software. Middleware-based load balancing services handle load balancing at the middleware level. Finally, application level load balancing is performed by the application itself. All these levels may employ load balancing found in the previous layer when supplying load balancing at a given layer. For instance, middleware level load balancing may employ load balancing facilities supplied by the operating system.

While load balancing can certainly performed in all these layers, some may have disadvantages that make them unsuitable for use in distributed applications that require dynamic adjustment to existing and future load conditions. Three prominent disadvantages include (1) the inability to take into account client request content, (2) lack of transparency, and (3) maintainability. In particular, network and operating system based load balancing suffer from the first disadvantage, *i.e.* they cannot take into account client request content because that information is application-specific. Application based load balancing suffers from the last two disadvantages. Transparency is lost since the application itself must be modified to support load balancing, which also introduces code maintenance issues.

Given these deficiencies, a cost-effective way to address the listed application demands is to employ load balancing services based on distributed object computing *middleware*, such as CORBA [**?**] or Java RMI [**?**]. These load balancing services distribute client workload equitably among various back-end servers to obtain improved response times.

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 [**?**, **?**]. In this approach, a naming service returns a reference to a different object each time it is accessed by a client. Load balancing via a naming service only supports a *non-adaptive* form of load balancing, however, which limits its applicability to distributed systems with more complex load balancing requirements. Non-adaptive load balancing also reduces the potential for optimizing overall distributed system load since the behavior of load balanced applications cannot be altered dynamically.

In contrast, *adaptive* load balancing services can consider dynamic load conditions when making load balancing decisions, which yields the following benefits:

- Adaptive load balancing services can be used for a larger range of distributed systems since they need not be designed for a specific class of application.
- Since a single load balancing service can be used for many types of applications, the cost of developing a load balancing service for specific classes of applications can be avoided, thereby reducing deployment costs.
- It is possible to concentrate on the load balancing service in general, rather than a particular aspect geared solely to a specific class of application, which can improve the quality of optimizations used in the load balancing service over time.

However, first-generation adaptive middleware load balancing services [**?**, **?**, **?**, **?**] do not provide solutions for key dimensions of the problem space. In particular, they provide insufficient functionality to satisfy complex distributed applications with higher optimization requirements. In general, as the complexity of distributed applications grows, their load balancing requirements necessitate more advanced functionality, 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 this advanced functionality can impede distributed system scalability. This thesis presents solutions to these and other types of load balancing challenges that are needed to optimize complex distributed systems more effectively. It specifically focuses on a CORBA-based solution.

CORBA's rich set of features provides the means to realize an adaptive load balancing service. CORBA is an effective choice for distributed systems due to the inherent distribution and common heterogeneity of clients and servers written in different programming languages running on different hardware and software platforms. In this context, CORBA can simplify system implementation because it offers a language- and platform-neutral communication infrastructure. Moreover, it reduces development effort by offering higher level programming abstractions that shield application developers from distribution complexities, thereby allowing them to concentrate their efforts on application logic, such as stock trading business logic in the example given in Section **??**.

1.2 Background

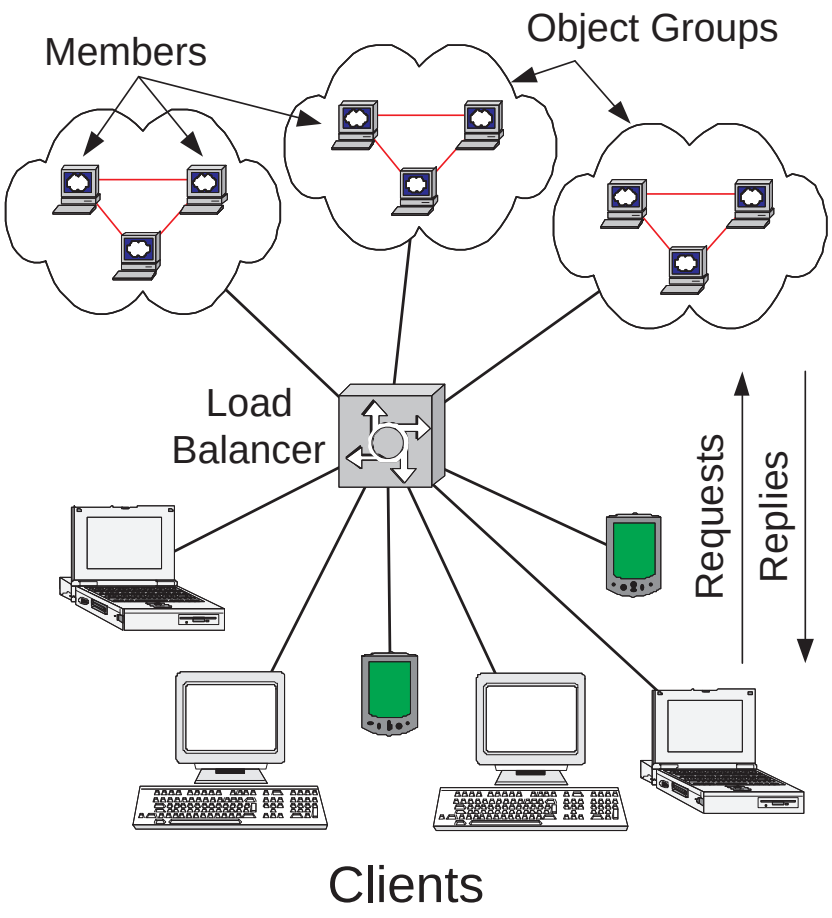

Figure 1.3: CORBA-based Load Balancing Concepts and Components

This thesis assumes that readers are familiar with the load balancing concepts and components[1] shown in Figure **??** and defined below:

- **Load balancer**, which attempts to ensure that application load is balanced across groups of servers. It is sometimes referred to as a "load balancing agent," or a "load balancing service." In this thesis, 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.
- **CORBA**, which is the OMG *Common Object Request Broker Architecture* [**?**] that defines interfaces, policies, and protocols that enable clients to invoke operations on distributed objects without concern

[1] The term *component* used throughout this paper refers to a "component" in the general sense, *i.e.*, an identifiable entity in a program, rather than in a more specific sense, *e.g.*, a component in the CORBA Component Model [**?**].

Research Challenge	*Solution Approach*	*Thesis Section*
Transparent server side load balancing support	Utilize Component Configurator to install Interceptors transparently	
Maximize throughput, minimize network and resource overhead	Lazy evaluation and queuing through AMI	

Table 1.1: Key Research Challenges

for object location, programming language, OS platform, communication protocols and interconnects, and hardware [**?**].

- **TAO**, which is *The ACE ORB* that provides an open-source[2] CORBA-compliant ORB designed to address applications with stringent quality of service (QoS) requirements.

The TAO CORBA ORB provides the distribution middleware for all of the components shown in Figure **??**. TAO facilitates location-transparent communication between:

- Clients and a load balancer
- A load balancer and the object group members and
- Clients and the object group members.

The load balancer also keeps track of which members belong to each object group. Section **??** provides more in-depth coverage of load balancing concepts and terminology.

1.3 Key Research Challenges

The key research challenges discussed in this thesis are listed in Table **??**. All challenges, their solutions, and research contributions are described in detail in subsequent chapters.

1.4 R&D Impact and Technology Transfer

Software research is often validated when its results are used in production applications. Much of the R&D described in this thesis has influenced load balancing concepts and work used in both academia and industry. In particular, notable impact of the R&D activities described in this thesis include the following:

- Established a set of load balancing concepts and nomenclature flexible and powerful enough to describe most load balancing applications and scenarios. Utilizing the above work, a load balancing model suitable for use in standards-based middleware was defined.
- The load balancing design and prototyping efforts that comprised this research were the primary contributors to the Object Management Group's forthcoming CORBA *Load Balancing and Monitoring* specification.
- Since this research forms the basis for an open standard, it will be reified and deployed in a wide range of commercial and academic middleware implementations, *i.e.*, CORBA ORBs. Thus far, two independent implementations of the forthcoming OMG CORBA Load Balancing and Monitoring specification have been developed for TAO and JacORB.

The technology transfer of the load balancing capabilities described in this theis further validate the importance of research results reported in this document.

[2]The software, documentation, examples, and tests for TAO and its adaptive load balancing service are open-source, and can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

1.5 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter ??** describes the key requirements that a CORBA-compliant load balancing service should be designed to address. It also qualitatively evaluates several alternative load balancing architectures that can be used for CORBA applications.
- **Chapter ??** discusses several aspects of the forthcoming OMG *Load Balancing and Monitoring* specification, including (1) the load balancing model, (2) the components of the load balancing service, and (3) the load balancing service semantics.
- **Chapter ??** provides an in-depth discussion of the design of TAO's standards-based CORBA load balancing service implementation.
- **Chapter ??** presents benchmarks of TAO's load balancing service that quantitatively evaluate how a CORBA-based load balancing service can improve overall throughput.
- **Chapter ??** describes other R&D efforts that are related to load balancing.
- **Chapter ??** summarizes future research on middleware-based load balancing in general and TAO's TAO's CORBA-compliant load balancing service in particular.
- **Chapter ??** presents concluding remarks.
- **Chapter ??** discusses how and where load balancing can be performed at several platform/application layers.
- **Chapter ??** briefly describes and lists the consolidated CORBA IDL defined by the forthcoming CORBA Load Balancing and Monitoring specification.

1.6 Summary

Existing middleware load balancing services often lack the flexibility and functionality needed to efficiently support a broad range of distributed applications that possess different usage patterns and resource requirements. Historically, middleware load balancing services are tailored for a specific class of distributed application. Such designs incur maintenance difficulties and unnecessary development time and costs to port to other classes of applications.

This chapter outlined key research challenges designed to address the limitations with existing middleware load balancing services. Solutions to these limitations have been reified in TAO's CORBA-based load balancing service, which forms the basis for the forthcoming OMG CORBA Load Balancing and Monitoring specification. Each of those solutions is discussed in subsequent chapters of this thesis.

Chapter 2

Load Balancing Requirements and Architectural Alternatives

This chapter describes the types of requirements that a CORBA-compliant load balancing service should be designed to address. It then qualitatively evaluates several alternative load balancing architectures that are suitable for CORBA-based applications.

2.1 Requirements for Middleware Load Balancing

2.2 Requirements for a CORBA Load Balancing Service

The OMG CORBA specification provides the core capabilities needed to support load balancing. In particular, a CORBA load balancing service can take full advantage of the standard CORBA *request forwarding* mechanism (described in Sidebar **??**) to forward client requests to other servers *transparently*, *portably*, and *interoperably*. The CORBA specification does not *standardize* load balancing interfaces, however. Nor does it specify load balancing mechanisms, which are left as implementation decisions for ORB providers. In the remainder of this section we therefore describe the key requirements that a CORBA load balancing service should be designed to address.

Sidebar 1: Overview of CORBA Request Forwarding

A servant can throw a `ForwardRequest` exception initialized with a copy of that reference. The server ORB catches this exception and then returns a LOCATION_FORWARD GIOP reply message to the client ORB. When the client ORB receives this message, the CORBA specification requires it to

1. Re-issue the request to the new location specified by the object references embedded in the LOCATION_FORWARD response and
2. To continue using that location until either the communication fails or the client is redirected again.

Support an object-oriented load balancing model. In the CORBA programming model objects are the unit of abstraction and system architects reason about objects in order to manage their available resources. Thus, the granularity of load balancing in CORBA should be based on objects, rather than, *e.g.*, processes or

TCP/IP addresses. Moreover, a load balancing service and ORB should coordinate the interactions amongst *multiple* replicas. Sets of multiple object replicas are called *object groups* or *replica groups*.

Client application transparency. Distributing work load amongst multiple servers should require little or no modifications to the way in which CORBA applications are developed normally. In particular, a CORBA load balancing service should be as transparent as possible to clients and servers. Likewise, a general principle in CORBA is that client implementations should be as simple as possible. A CORBA load balancing service that follows this principle should therefore require no changes to clients whose requests it balances.

Server application transparency. Although load balancing should ideally require few modifications to servers, this goal is hard to achieve in practice. For example, load balancing a stateful CORBA object requires the transfer of its state to a new replica. The application implementation must either perform the transfer itself or define hooks that allow the load balancing framework to perform the state transfer as unobtrusively as possible [**?**].

The situation for stateless CORBA servers is different. In this case, the implementation of an server object's *servant*[1] should require no changes to support load balancing. Yet changes to the server *application* may still be required under certain conditions. For example, some applications may define *ad hoc* load metrics, such as number of active transactions or user sessions. In practice, collecting these metrics may require some modifications to server application code.

Dynamic client operation request patterns. Load balancing services can be based on various client request patterns. For example, load balancers for certain types of systems assume client requests occur at deterministic or stochastic rates that execute for known or fixed durations of time. While these assumptions may apply for certain types of applications, such as continuous multimedia streaming [**?**], they do not apply in complex Internet or military [**?**] environments where client operation request patterns are dynamic and the duration of each request may not be known in advance. In this paper, therefore, we focus on load balancing techniques that do not require *a priori* scheduling information.

Maximize scalability and equalize dynamic load distribution. Although it is common practice to design lightweight load distribution capabilities, *e.g.*, based on extensions to naming services [**?**], these approaches do not balance dynamic loads equitably, which limits their scalability. Thus, a CORBA load balancing service must increase system scalability by maximizing dynamic resource utilization in a group of servers whose resources would not otherwise be used as efficiently. By improving resource utilization via load balancing, the overall scalability of the server group should be enhanced significantly.

Support administrative tasks. System administrators may need to add new object replicas dynamically, without disrupting or suspending service for existing clients. A good CORBA load balancing service should allow the dynamic addition of new replicas and adjust to the new load conditions rapidly. Likewise, the service should allow the removal of replicas for upgrades, preemptive maintenance, or re-allocation of system resources.

Minimal overhead. A CORBA load balancing service should not introduce undue latency or networking overhead since otherwise it can actually reduce–rather than enhance–overall system performance. In particular, an implementation that (1) increases the average number of messages per-request or (2) uses a single server to process all requests may be inappropriate for high-performance and/or large-scale applications. Chapter **??** illustrates empirically how certain load balancing strategies can degrade overall performance due to excess overhead.

Support application-defined load metrics and balancing policies. Different types of applications have different notions of load. Thus, a CORBA load balancing service should allow applications to:

- *Specify the semantics of metrics used to measure load* – For example, some applications may want to balance CPU load, whereas other applications may be more concerned with balancing I/O resources, communication bandwidth, or memory load.

[1] The servant is a programming language entity that implements object functionality in a server application.

Figure 2.1: CORBA Load Balancing Strategies

- *Set policies that determine the load balancing service's semantics* – For example, some applications may want to distribute load uniformly, others randomly, and still others may want load distributed based on dynamic metrics, such as current CPU load or current time.

Support for application-defined metrics and policies need not affect client transparency because these policies can be administered solely for server replicas. Clients can therefore be shielded from knowledge of load balancing metrics and policies.

CORBA interoperability and portability. Application developers rarely want to be restricted to a single provider's ORB. Therefore, a CORBA load balancing service should not rely on extensions to GIOP/IIOP, which are standard protocols that allow heterogeneous CORBA clients and servers to interoperate. Likewise, it is desirable to avoid implementing load balanced objects by adding proprietary extensions to an ORB.

2.3 Alternative CORBA Load Balancing Strategies and Architectures

There are a variety of strategies and architectures for devising CORBA load balancing services. Different alternatives provide different levels of support for the requirements outlined in Sections **??** and **??**, as described below.

2.3.1 Load Balancing Strategies

There are various strategies for designing CORBA load balancing services. These strategies can be classified along the orthogonal dimensions shown in Figure **??** and discussed below:

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

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

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

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

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

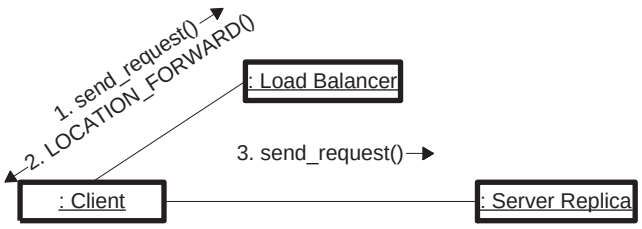

Figure 2.2: A Non-Adaptive Per-Session Architecture

2.3.2 Load Balancing Architectures

By combining the strategies shown in Figure **??** and described in Section **??** in various ways, it is possible to create the alternative load balancing architectures described below. In the ensuing discussion, we refer to the requirements presented in Sections **??** and **??** to evaluate the pros and cons of these strategies *qualitatively*. Chapter **??** then evaluates these different strategies *quantitatively*.

Non-adaptive per-session architectures. One way to design a CORBA load balancer is make to the load balancer select the target replica when a client/server session is first established, *i.e.*, when a client obtains an object reference to a CORBA object–namely the replica–and connects to that object, as shown in Figure **??**. Note that the balancing policy in this architecture is *non-adaptive* since the client interacts with the same server to which it was directed originally, regardless of that server's load conditions. This architecture is suitable for load balancing policies that implement round-robin or randomized balancing algorithms.

Load balancing services based on a per-session client binding architecture can be implemented to support many of the requirements defined in Sections **??** and **??**. For example, per-session client binding architectures generally satisfy requirements for application transparency, minimal overhead, and CORBA interoperability. The primary benefit of per-session client binding is that it incurs less run-time overhead than the alternative architectures described below.

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

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

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

Adaptive per-session architecture. This architecture is similar to the non-adaptive per-session approach. The primary difference is that an adaptive per-session can use run-time load information to select the replica, thereby alleviating the need to bind new clients to heavily loaded replicas. This strategy only represents a

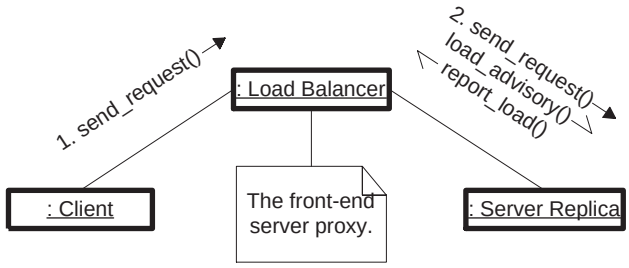

Figure 2.3: An Adaptive Per-request Architecture

slight improvement, however, since the load generated by clients can change after binding decisions are made. In this situation, the adaptive on-demand architecture offers a clear advantage since it can respond to dynamic changes in client load.

Adaptive per-request architectures. A more adaptive request architecture for CORBA load balancing is shown in Figure **??**. This design introduces a front-end server, which is a proxy [**?**] that receives all client requests. In this case, the "front-end server" is the load balancer. The load balancer selects an appropriate back-end server replica in accordance with its load balancing policy and forwards the request to that replica. The front-end server proxy waits for the replica's reply to arrive and then returns it to the client. Informational messages–called *load advisories*–are sent from the load balancer to replicas when attempting to balance loads. These advisories cause the replicas to either accept requests or redirect them back to the load balancer.

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

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

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

Adaptive on-demand architecture. As shown in Figure **??**, clients receive an object reference to the load balancer initially. Using CORBA's standard LOCATION_FORWARD mechanism, the load balancer can redirect the initial client request to the appropriate target server replica. CORBA clients will continue to use the new object reference obtained as part of the LOCATION_FORWARD message to communicate with this replica directly until they are either redirected again or they finish their conversation.

Unlike the non-adaptive architectures described earlier, adaptive load balancers that forward requests on-demand can monitor replica load continuously. Using this load information and the policies specified by an application, a load balancer can determine how equitably the load is distributed. When load becomes unbalanced, the load balancer can communicate with one or more replicas and request them to redirect subsequent clients back to the load balancer. The load balancer will then redirect the client to a less loaded replica.

Using this architecture, the overall distributed object computing system can (1) recover from unequitable client/replica bindings while (2) amortizing the additional network and processing overhead over multiple requests. The remainder of this paper focuses primarily on this architecture since it addresses most

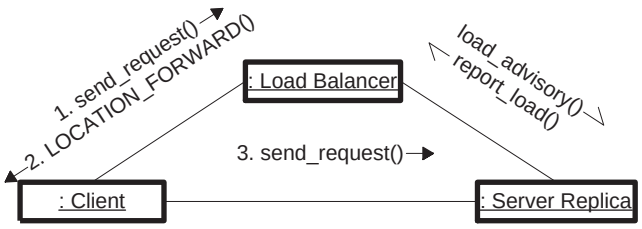

Figure 2.4: An Adaptive On-Demand Architecture

of the requirements in Sections **??** and **??**. In particular, it requires minimal changes to the application initialization code and no changes to the object implementations (servants) themselves.

The primary drawback with adaptive on-demand architectures is that object implementations must be prepared to receive messages from a load balancer and redirect clients to that load balancer. Although the required changes do not affect application logic, application developers must modify a server's initialization and activation components to respond to the load advisory messages mentioned above. Advanced ways of overcoming this drawback are discussed in Section **??**.

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

2.4 Summary

Chapter 3

The Forthcoming OMG Load Balancing Service Architecture

This chapter describes the architecture of the load balancing service defined in the forthcoming OMG *Load Balancing and Monitoring* specification, which is based on the research described in this thesis. Of particular interest is how the forthcoming OMG architecture addresses several important challenges when designing and using load balancing services.

3.1 Basic Model

The basic model employed by the forthcoming OMG Load Balancing and Monitoring architecture is location-oriented, as opposed to process-oriented or object-oriented. In the non-adaptive load balancing case described in Section **??**, the member to receive the next client request is based on the *location* where a specific member of an object group resides. The adaptive 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.

Typically, hosts or "nodes" are associated with locations. However, the Model makes no assumptions about the application's interpretation of what a "location" is. For example, an application could decide to associate a CORBA object with a location instead of the host. Note that the load balancing model is still location-oriented in this case since a load balancer would be oblivious to the fact that the location is actually a process.

The deployed structure of the location-oriented load balancing service is shown in Figure **??**. The Model allows for members from different object groups to reside at the same location. For instance, `member`

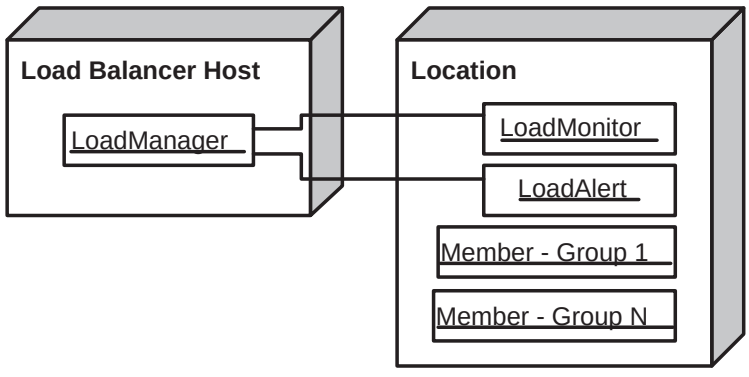

Figure 3.1: Deployed Structure of the Location-Oriented CORBA Load Balancing Service.

2 from `group 1` and `member 7` from `group 2` can each reside at `location 1`. This flexibility is one of the strengths of the Model. Additional flexibility can be found in the Model's support for object group-specific properties, such as the load balancing strategy in use.

A more detailed description and break-down of the components shown in Figure **??** follow.

3.2 Component Structure in the OMG Load Balancing and Monitoring Service

Figure **??** illustrates the relationships among the components in the OMG Load Balancing and Monitoring specification.

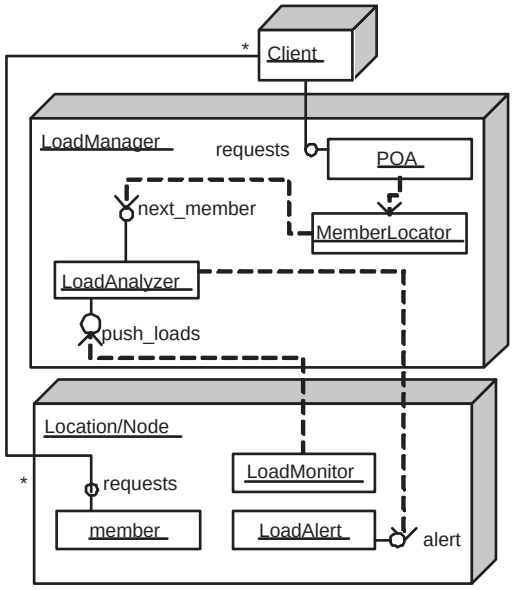

Figure 3.2: Components in the CORBA Load Balancing and Monitoring Service

The preceding discussion outlines the elements of the forthcoming CORBA Load Balancing and Monitoring specification but does not motivate what these elements do or more importantly *why* they are important. In the remainder of this section, we explain why these elements are needed by explaining the key challenges they address, which include:

1.

3.2.1 Context.

Problem.

Solution → Load manager. Define a load manager component that integrates all the other components shown in Figure **??**. The load manager component is a mediator [**?**] that provides an interface through which load balancing can be administered, without exposing clients to the intricate interactions between the components it integrates. The term "load balancer" refers to all components that logically comprise the load balancer. "Load manager," on the other hand, is one only component found within the logical load balancer entity.

3.2.2 Context.

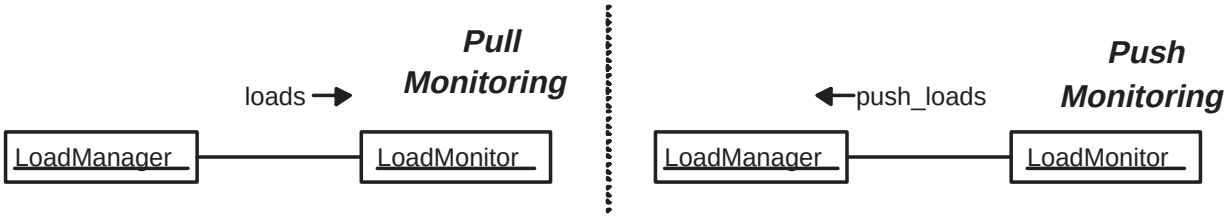

Figure 3.3: Load Reporting Policies

Problem.

Solution → Member locator. Define a member locator component that identifies which group members will receive which requests. This component is also the mechanism that binds clients to the identified members. A member locator can be implemented portably using standard CORBA portable object adapter (POA) mechanisms, such as servant locators [**?**], which are a standard part of all CORBA-compliant ORBs. Servant locators are themselves implementations of the Interceptor pattern [**?**], which ... The member locator forwards each request it receives to the member selected by the load analyzer described below.

3.2.3

Context.

Problem.

Solution → Load analyzer. Define a load analzer component that decides which member will receive the next client request. The member locator described in Section **??** obtains a reference to a member from the load analyzer and then forwards the request to that member. The load analyzer also allows a load balancing strategy to be selected explicitly at run-time, while 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.

3.2.4

Context.

Problem.

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

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

3.2.5

Context.

Problem.

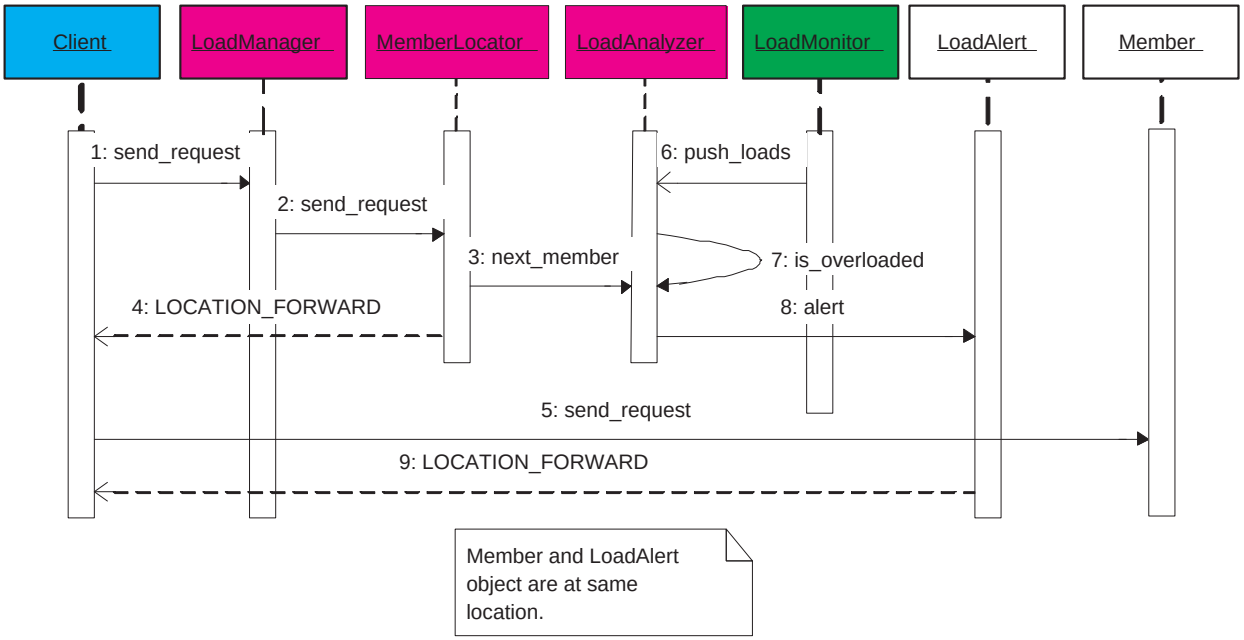

Figure 3.4: OMG Load Balancing and Monitoring Interactions

Solution → Load alert. This component facilitates load shedding. It responds to *alert* conditions set by the load analyzer component described above. If the load analyzer requires that load be shed from the location a given object group member resides at, it will enable an "alert" condition on the load alert object residing at that same location. Once the alert is enabled, the load alert object reject client requests. Clients will be transparently forward back to the load balancer for reassignment to another member.

3.3 Dynamic Interactions in the OMG Load Balancing and Monitoring Service

As described in Section **??**, 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, the OMG Load Balancing and Monitoring service architecture therefore provides a hybrid solution, whose interactions are shown in Figure **??**. Each interaction in Figure **??** is outlined below.

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 at. The additional indirection and overhead incurred by per-request load balancing architectures (see Section **??**) 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 **??**) that is configured, the load monitor will either report the load(s) to the load analyzer (via the load manager) or the load manager will query the load monitor for the load(s) at a given location.

7. As loads are collected by the load manager, the load analyzer analyzes the load at all known locations.
8. To fulfill the transparency requirements outlined in Section **??**, 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 the next request sent by a client back to the load manager.

After all these steps, the load balancing cycle begins again.

3.4 Design Challenges and Their Solutions

The following design challenges were identified prior to and during the development of the TAO load balancer prototype that drove the model, architecture and content of the forthcoming OMG Load Balancing and Monitoring specification:

1. Implementing portable load balancing
2. Enhancing feedback and control
3. Supporting modular load balancing strategies
4. Coping with adaptive load balancing hazards
5. On-demand member activation
6. Integrating all the load balancing components effectively

The challenges and the solutions that were applied to address them are discussed below. The solutions to each design challenge manifest themselves within the load balancing service components described in Section **??**. Readers who are not interested in the design and rationale of TAO's load balancing service, and hence the OMG Load Balancing and Monitoring specification, should skip to the performance results in Chapter **??**.

3.4.1 Challenge 1: Implementing Portable Load Balancing

Context. A CORBA load balancing service is being implemented in accordance with the requirements outlined in Chapter **??**.

Problem. Changing application code–particularly client applications–to support load balancing can be tedious, error-prone, and costly. Changing the middleware infrastructure to support load balancing is also problematic since the same middleware may be used in applications that do not require load balancing, in which case extra overhead and footprint may be unacceptable. Likewise, using *ad hoc* or proprietary interfaces to add load balancing to existing middleware can increase maintenance effort and may be unattractive to application developers who fear "vendor lock-in" from features that are unavailable in other middleware.

So, how can we implement load balancing transparently *without* changing applications, middleware or using proprietary features?

Solution → the Interceptor pattern. The Interceptor pattern [**?**] allows a framework to transparently add services that are triggered automatically when certain events occur. This pattern enhances extensibility by exposing a common interface implemented by a *concrete interceptor*. Methods in this interface are invoked by a *dispatcher*.

The Interceptor pattern can be implemented via standard CORBA POA [**?**] features. For example, the role of the interceptor is played by a *servant locator*[1] and the role of the dispatcher is played by a *POA*.

[1] Servant locators are a meta-programming mechanism [**?**] that allows CORBA server application developers to obtain custom object implementations dynamically, rather than using the POA's active object map [**?**].

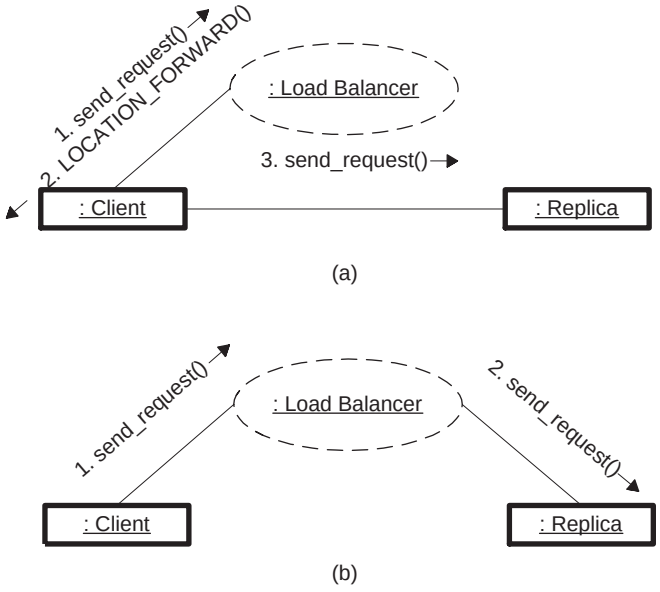

Figure 3.5: Load Balancing Transparency in Applications: (a) request forwarded by the client and (b) request forwarded on behalf of the client.

In particular, a *member locator* can implement the standard CORBA `ServantLocator` [**?**] interface provided by the POA.

Figure **??** illustrates how load can be balanced transparently using standard CORBA features. Initially, clients are given an object reference to the load balancer (actually the load manager), so they first issue requests to the load balancer. The load balancer's servant locator intercepts those requests and forwards them transparently to the appropriate members. Depending on the type of client binding granularity (see Section **??**) selected by the application, one of the following actions will occur:

- The client will forward requests to the appropriate member, as shown in Figure **??**(a); or
- The load balancer will forward requests to the appropriate member on behalf of the client, as shown in Figure **??**(b).

Applying the solution. When using an OMG Load Balancing and Monitoring service implementation, such as the one found in TAO, each member registers itself with an object group managed by the load balancer. Each member then becomes a potential candidate to handle a request intercepted by the load balancer. The interception is performed by a servant locator.

The load balancer implements its own servant locator, which is registered with the load balancer's POA. When a new request arrives, the POA delegates the task of locating a suitable servant to the servant locator, rather than using the servant lookup mechanism in the POA's active object map [**?**]. Thus, the load balancer can use the servant locator to forward requests to the appropriate member transparently, *i.e.*, without affecting server application code.

After receiving a request, the member locator obtains a reference to the member chosen by the load analyzer (see Section **??**) and throws a `ForwardRequest` exception initialized with a copy of that reference. The server ORB catches this exception and then returns a LOCATION_FORWARD GIOP reply message, which is described in Sidebar **??** (page **??**). A server application and an ORB can therefore forward client requests to other servers *transparently*, *portably*, and *interoperably*.

3.4.2 Challenge 2: Enhancing Feedback and Control

Context. An *adaptive* load balancing service must determine the current load conditions on members registered with it. A load balancer should not need to know the type of load metric beforehand, however. Moreover, a load balancer must take steps to ensure that loads across its registered members are balanced. These steps include (1) forcing the member to redirect the client back to the load balancer when its load is high and (2) forcing the member to once again accept client requests when its load is nominal.

Problem. Sampling loads from locations should be as transparent as possible to the application. If load sampling was not transparent, a load balancer would have to sample loads directly from member, which is undesirable since it would require members to collect loads. If member collect loads, however, application developers must modify existing application code to support load balancing. Such an obtrusive design does not scale well from a deployment point of view, nor is it always feasible to alter existing application code.

Moreover, a load balancer should not be tightly coupled to a particular load metric. Only the *magnitude* of the load should be considered when making load balancing decisions, so that a load balancer can support any type of load metric, rather than just one type of metric. The same deployment scalability issues encountered for load sampling transparency also apply here. If a load balancer were load-metric specific it would be costly to deploy load balancers for distributed applications that require balancing based on several load metrics. For example, a separate load balancer would be needed to balance members based on various metrics, such as CPU, I/O, memory, network, and battery power utilization.

In addition, 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 subsequent requests can be reassigned to a less loaded member.

So, how can we implement a flexible load balancing service that can be extended to support new load metrics, as well as different policies to collect such metrics?

Solution → the Strategy and Mediator patterns. The Strategy [**?**] design pattern allows the behavior of frameworks and components to be selected and changed flexibly. For example, the same interface can be used to obtain different types of loads on a given set of resources. Only object implementations must change since load measuring techniques may differ for each type of load. Each implementation is called a "strategy" and can be embodied in an object called a *load monitor*.

A load monitor implements a strategy for monitoring loads on a given resource. The interface for reporting loads to the load balancer or to obtain loads from the load monitor remains unchanged for each load monitoring strategy. Strategizing load monitoring makes it possible to use a load balancer that is not specific to a particular type of load, such as CPU load or battery power utilization. Thus, a load balancer need not be specialized for a given type of load. This design simplifies deployment of a load balanced distributed system since one load balancer can balance many different types of load.

The Mediator [**?**] design pattern defines an object that encapsulates how objects will interact. A load alert object acts as a mediator between the load balancer and a given member. This pattern ensures there is a loose coupling between the load balancer and the group members. Thus, the load balancer need not have any knowledge of the interface exported by the member.

A load alert object responds to load balancing requests sent by the load balancer. Depending on the type of request the load balancer sends to the load monitor, the member will either be forced continue accepting client requests or redirect the client back to the load balancer. Note that the load balancer never interacts with the member directly – all interaction occurs via the load alert object. Similarly, the member never interacts with the load balancer directly. This is depicted in Figure **??**.

The `ServiceContexList` shown in this figure is "out-of-band" data that is transparently sent along with the client's request. It is used to identify the target of the request as a load balanced one. It is necessary to send this out-of-band data to identify load balanced targets since not all targets, *i.e.*, CORBA objects residing at a given location may be load balanced. In those cases, the load shedding mechanism, the load alert object, should not attempt to control those requests.

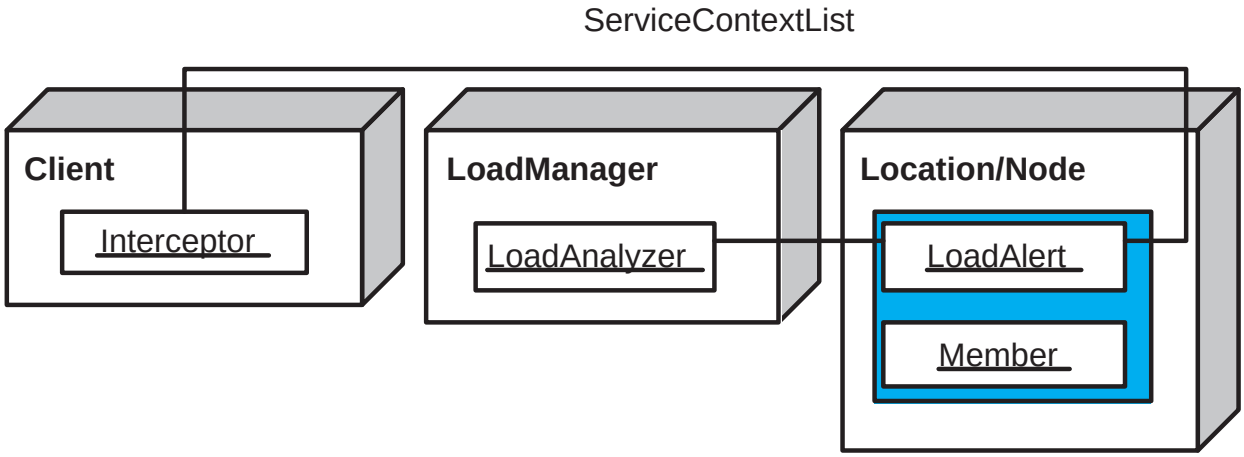

Figure 3.6: The `LoadAlert` Mediator

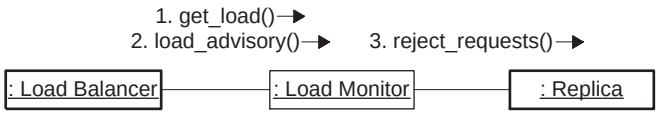

Figure 3.7: Feedback and Control When Balancing Loads

Applying the solution. When enabling adaptive load balancing in a particular distributed application, a load monitor (in the "pull" monitoring case) and a load alert object are registered with the load balancer. As shown in Figure **??**, the load balancer 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 **??**). 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 object from the load balancer and set the "alert" state 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 object 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 **??**.
This figure shows two additional entities not previously discussed. They are:

- the `ClientRequestInterceptor`;
- the `ServerRequestInterceptor`.

They expose a standard CORBA interface, and are really used in the figure to illustrate how to transparently and portably provide load shedding functionality. Descriptions of how they are used in the overall load shedding interactions follow:

1. A client request is intercepted by the `ClientRequestInterceptor`.

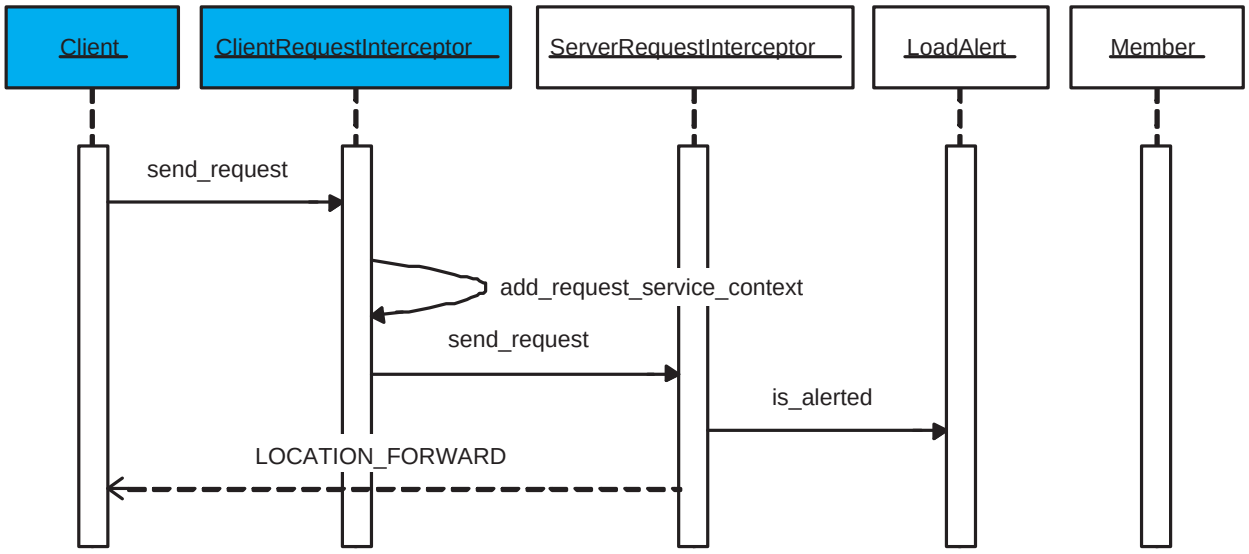

Figure 3.8: Load Shedding Interactions

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 `ServiceContextList` for the identification information injected on the client side, and if the load alert object has been told that it should reject requests.

5. If the information exists and requests are to be rejected, the `ServerRequestInterceptor` will issue the appropriate exception to force the client to re-invoke its request on the load balancer.

Armed with a load monitor and load alert object, such a load balancer is *adaptive* due to the bi-directional feedback/control channel between the load monitor, load alert object 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.

3.4.3 Challenge 3: 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.

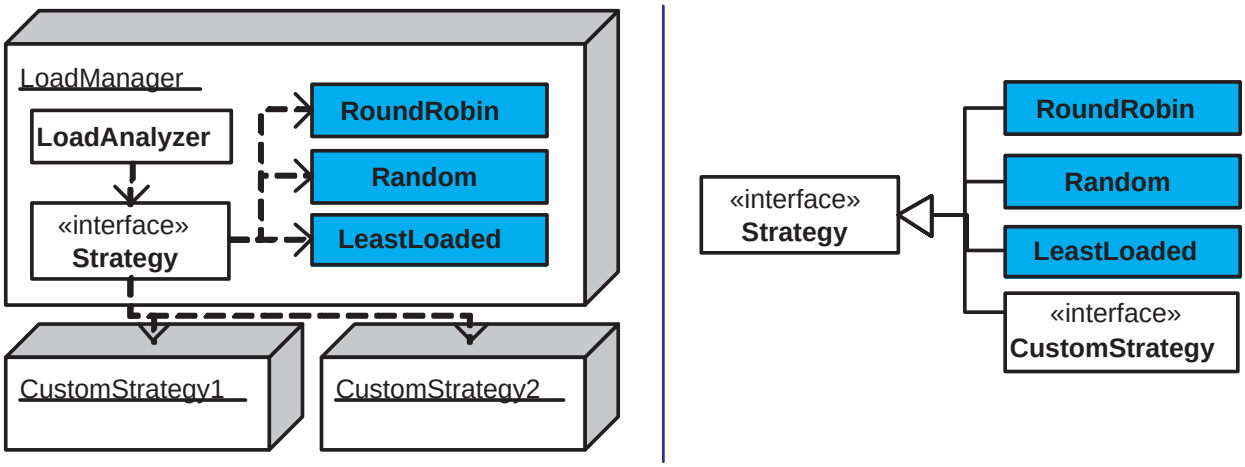

Figure 3.9: Applying the Strategy Pattern to the OMG Load Balancing Service

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 24×7 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 OMG load balancing service this pattern can be used to change the member selection strategy dynamically. Thus, a load balancer can 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. The load analyzer uses the Strategy pattern to customize the load balancing algorithm used when making load balancing decisions, as depicted in Figure **??**. The OMG load balancing service can be configured to 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 [**?**].

- **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.[2]

However, the OMG load balancer is not limited to these built-in strategies. 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 [**?**] 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.

3.4.4 Challenge 4: Coping With Adaptive Load Balancing Hazards

Context. A customized adaptive load balancing strategy is under development by a distributed application developer. This load balancing strategy will be used to balance loads across a group of replicas.

Problem. Adaptive load balancing has the potential to improve system responsiveness. It is hard to ensure the stability of loads across replicas when the overall state of distributed systems changes quickly due to the following hazards:

- **Thundering herd.** When a member at a less loaded location suddenly becomes available, a "thundering herd" phenomenon may occur if the load balancer forwards all requests to that member immediately. If the rate at which the loads are reported and analyzed is slower than the rate at which requests are forwarded to the member, it is possible that the load on that member will increase rapidly. Ideally, the rate at which requests are forwarded to member should be less than or equal to the rate at which loads are reported and analyzed. Satisfying this condition can eliminate the thundering herd phenomenon.

- **Balancing paroxysms.** The smaller the number of members, the harder it can be to balance loads across them effectively. For example, if only two members are available then one member may be more loaded than the other. A naive load balancing strategy will attempt to shift the load to the member at the less loaded location, at which point it will most likely become the member with the greater load. The entire process of shifting the load may begin again, causing system instability.

So, how can we adapt to dynamic changes in load, but without over reacting to transient, short lived or sample errors in the load metric?

Solution → Dampening load sampling rates and request redirection. The *least loaded* load balancing strategy described in Section **??** can be employed to alleviate the thundering herd phenomenon and balancing paroxysms since it will not attempt to shift loads the moment an imbalance occurs. Specifically, by relaxing the criteria used to decide when loads across a group of member is balanced, a load balancer can adjust to large load discrepancies with less probability of experiencing the hazards discussed above. The criteria for deciding when to shift loads can also change dynamically as the number of members increases.

Using control theory terminology, this behavior is called *dampening*, where the system minimizes unpredictable behavior by reacting slowly to changes and waiting for definite trends to minimize over-control decisions. When configured to use dampening, the least loaded balancing strategy does not react to changes in load immediately because it averages instantaneous load samples with older load values. The empirical results presented in Section **??** illustrate the effects of the dampening mechanism.

3.4.5 Challenge 5: On-demand Member Activation

Context A load balanced distributed application starts out with a given number of members. Depending on availability of resources, such as CPU load and network bandwidth, the number of member may need to grow or decrease over time.

[2] 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*.

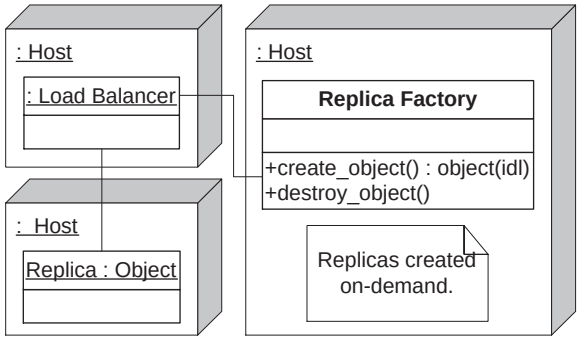

Figure 3.10: On-demand Member Creation

Problem The scenario presented above requires that replicas be created or destroyed on-demand. However, the load balancing service must have a means to create or destroy replicas.

Solution → the Factory pattern The Factory pattern [**?**] exposes an interface through which objects can be created. A load balancing service can use factory objects, *i.e.*, objects that implement the Factory design pattern, to create members on-demand. The load balancing service would simply invoke remote operations on the factory object at a given location when it decides that more replicas are necessary to maintain balanced loads.

For example, suppose there are only two members in an object group and that their loads are high. Without additional members, it may not be feasible to maintain balanced loads. A load balancing service with the ability to create and destroy replicas on-demand may provide more flexible load balancing strategies, *e.g.*, a load balancer can create a member at a third location to decrease the workload on the two initial members, as shown in Figure **??**.

On-demand creation and destruction of members allows resources to be used more efficiently. For example, starting a member before it is needed may impose additional resource utilization since the member must wait for requests to be sent to it. Depending on the member design and the middleware implementation, this "eager" allocation design can use a significant amount of resources, thereby reducing the amount of resources available to other processes running on the same host the unused member is running on. On-demand creation and destruction of members alleviates these problems.

Applying the solution. Those familiar with fault tolerance services may recognize a similarity between their member management strategies and those of load balancing services. Both types of services can control member lifetimes, *e.g.*, by creating members on-demand. A fault tolerance service requires sufficient members to provide fault recovery, while a load balancing service requires enough members to provide balanced loads. Although the underlying functionality for each type of service is different, the interface exposed by each service can be similar. Therefore, the IDL interfaces exposed by the forthcoming OMG Load Balancing and Monitoring specification is based largely on the IDL interfaces standardized by the Fault Tolerant CORBA specification [**?**].

3.4.6 Challenge 6: Integrating All the Load Balancing Components Effectively

Context. As illustrated above, a load balanced distributed system has many components that interact with each other. For example, clients issue requests to members. Load monitors measure loads on locations continuously. Load alert objects control client access to the members. Load analyzers decide if loads on locations are nominal or high. Finally, member locators bind clients to members.

Problem. All the components mentioned above must collaborate effectively to ensure that a distributed system is load balanced. Direct interaction between some of those components may complicate the im-

plementation of distributed applications, however, since certain functionality may be exposed to a given component unnecessarily.

So, how can we integrate the functionality of all the load balancing components without unduly coupling all of them?

Solution → the Mediator pattern. The Mediator pattern provides a means to coordinate and simplify interactions between associated objects. This pattern shields the objects from relationships and interactions that are not needed for their effective operation.

A *load balancer* component can be used to tie together all the components listed above. It coordinates all interactions between other components, *i.e.*, it is a mediator. Thus, clients can remain unaware of the interactions mediated by the load balancer, which helps to satisfy application transparency requirements.

Applying the solution. As shown in Figure **??**, the load balancer mediates the following types of component interactions:

• **Client binding interactions.** Rather than binding itself to a specific member at a location that may be highly loaded, the load balancer binds the client to a suitable member. The load balancer creates an object reference that corresponds to a group of members–called an *object group*–being load balanced. Instead of using an object reference that directly refers to a given member, the client uses the object reference created by the load balancer that represents the appropriate object group. This design causes the client to invoke a request on the load balancer initially, at which point the client is re-bound to a member chosen by the load balancer. The load balancer also rebinds the client to another member by using other components, such as the load alert object. In that case, a client is forwarded back to the load balancer so that the client binding process can be begin again. Thus, load balancing remains completely transparent to client applications.

• **Load monitor and load analyzer interactions.** The load balancer allows the load analyzer to be completely decoupled from load monitors. Load monitors are registered with the load balancer (in the "pull" monitoring case). This design allows the load balancer to receive load reports from each registered load monitor. These load reports are then delegated to the load analyzer for analysis. The means by which these loads were obtained is hidden from the load analyzer.

3.5 Summary

The forthcoming OMG Load Balancing and Monitoring specification is an important step toward greatly improving the scalability of CORBA-based distributed applications. Its design tackles the key requirements of middleware and CORBA load balancing services described in Sections **??** and **??**. Moreover, it defines a robust model and architecture, powerful and flexible enough to provide a great deal of load balancing functionality in a transparent, interoperable, standard and portable way. The OMG Load Balancing and Monitoring specification is heavily influenced by the research on TAO's XZY efforts presented in this thesis.

Chapter 4

The Design of the TAO CORBA Load Balancing Service

This chapter describes the design of the XYZ adaptive load balancing service in TAO [**?**], which is a CORBA-compliant ORB that supports applications with stringent QoS requirements. TAO's XYZ 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. This is all done portably due to the fact that TAO's XYZ load balancing service implements the forthcoming standard OMG Load Balancing and Monitoring specification.

4.1 Design Challenges and Their Solutions

In addition to the design challenges set forth in Section **??**, several TAO load balancer-specific design challenges were established during its design and implementation. They are:

1. Complete server transparency
2. Maximizing throughput and minimizing network and resource overhead

4.2 Challenge 1: 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 → 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 [**?**]. These mechanisms can alter the behavior of a client or a server when processing a

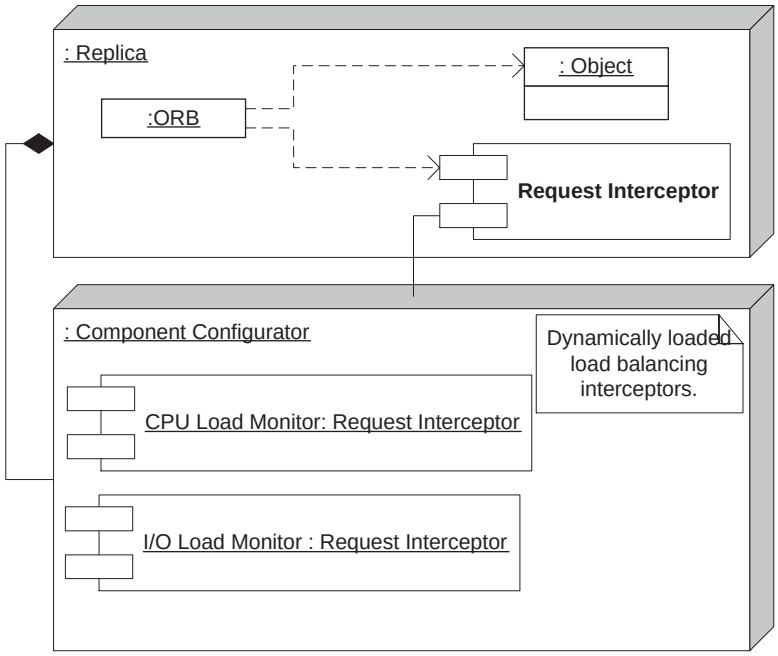

Figure 4.1: Transparent Server-side Load Balancing

given client request [**?**]. 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 [**?**] 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 **??**. Using this approach, the overall throughput of a distributed application can be improved without modifications to distributed application server code.

Applying the Solution in TAO 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:

- Portable interceptors: Portable interceptors [**?**] 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 [**?**] introduces *containers* to decouple application component logic from the configuration, initialization, and administration of servers. In the CCM, a container creates the POA[1] 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 balanc-

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

ing service. The standard CCM containers can be extended to implement automatic load balancing *generically* without changing application component behavior.

4.3 Challenge 2: 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 **??**.

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.

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` object described in Section **??**.

One technique to avoid this delay is to use non-blocking CORBA *one way* invocations. However, such invocations are not guaranteed to arrive at the intended target, nor is it possible to convey exceptional conditions back to the load balancer. The ability to determine the health of the remote `LoadAlert` object 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). 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 TAO Applying lazy evaluation to TAO's load balancer was relatively straightforward. Load analysis, member selection and load shedding functions were simply not called until a client made an invocation on the load balancer. Once the member locator is invoked, load analysis, member selection and load shedding begin.

Incorporating AMI into the remote load shedding invocations was also straightforward. A reply handler was 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[2] parameter passed to them.

4.4 Summary

The forthcoming OMG Load Balancing and Monitoring specification defines a powerful model and architecture. It does not, however, dictate how specific load balancing services should be implemented, nor should it. It is up to the load balancing service implementor to determine how best to tune the implementation for optimal performance and functionality.

In TAO's case, two issues not addressed by the OMG Load Balancing and Monitoring specification were handled. First, the interface provided by the Specification is very useful but it is very tedious to use. Furthermore, it requires that servers be modified to make calls on the load balancer. This greatly reduces server side transparency. TAO addresses this deficiency by using a Component Configurator implementation to dynamically load all code required to add load balancing support to a server, thus obviating the need to modify the server code.

Second, a naive implementation of the Specification would have inherent inefficiencies that reduce its overall performance and increase its resource utilization. A hybrid *lazy evaluation*/*AMI* approach overcomes such inefficiencies.

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

Chapter 5

Empirical Results

For load balancing to improve the overall performance of CORBA-based systems significantly, the load balancing service must incur minimal overhead. A key contribution of TAO's YXZ load balancing service is its ability to increase overall system throughput by distributing requests across multiple back-end servers (object grou members), without increasing round-trip latency and jitter significantly. This section describes the design and results of several experiments we performed to measure the benefits of TAO's load balancing strategy empirically, as well as to demonstrate the limitations with the alternative load balancing strategies outlined in Section **??**.

Section **??** outlines the hardware and software platform used to benchmark XYZ. The first set of experiments in Section **??** show the amount of overhead incurred by the request forwarding architectures described in this paper. The second set of experiments in Section **??** demonstrate how TAO's load balancer can maintain balanced loads dynamically *and* efficiently, whereas alternative load balancing strategies cannot.

5.1 Hardware/Software Benchmarking Platform

Benchmarks performed for this paper were run using three 733 MHz dual CPU Intel Pentium III workstations, and one 400 MHz quad CPU Intel Pentium II Xeon workstation, all running Debian GNU/Linux "potato" (GLIBC 2.1), with Linux kernel version 2.2.16. GNU/Linux is an open-source operating system that supports kernel-level multi-tasking, multi-threading, and symmetric multiprocessing. All workstations are connected through a 100 Mbps ethernet switch. This testbed is depicted in Figure **??**. All benchmarks were run in the POSIX real-time thread scheduling class [**?**]. This scheduling class enhances the integrity of our results by ensuring the threads created during the experiment were not preempted arbitrarily during their execution.

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

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

2. Latency test with non-adaptive per-request load balancing strategy: This variant of `Latency` test was designed to demonstrate the performance and scalability of *optimal* load balancing using per-request forwarding as the underlying request forwarding architecture. This variant added a specialized "forwarding

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

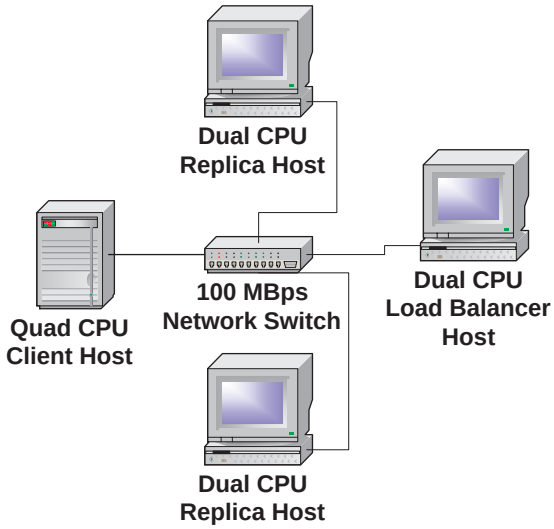

Figure 5.1: Load Balancing Experiment Testbed

server" to the test, whose sole purpose was to forward requests to a target server at the fastest possible rate. No changes were made to the client.

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

5.2 Benchmarking the Overhead of Load Balancing Mechanisms

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

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

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

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

3. A non-adaptive per-request client binding architecture: Next, we added a specialized non-adaptive per-request "forwarding server" to the original `Latency` test. This server just forwards client requests to an unmodified backend server. The forwarding server resided on a different machine than either the client or backend server, which themselves each ran on separate workstations. Since the forwarding

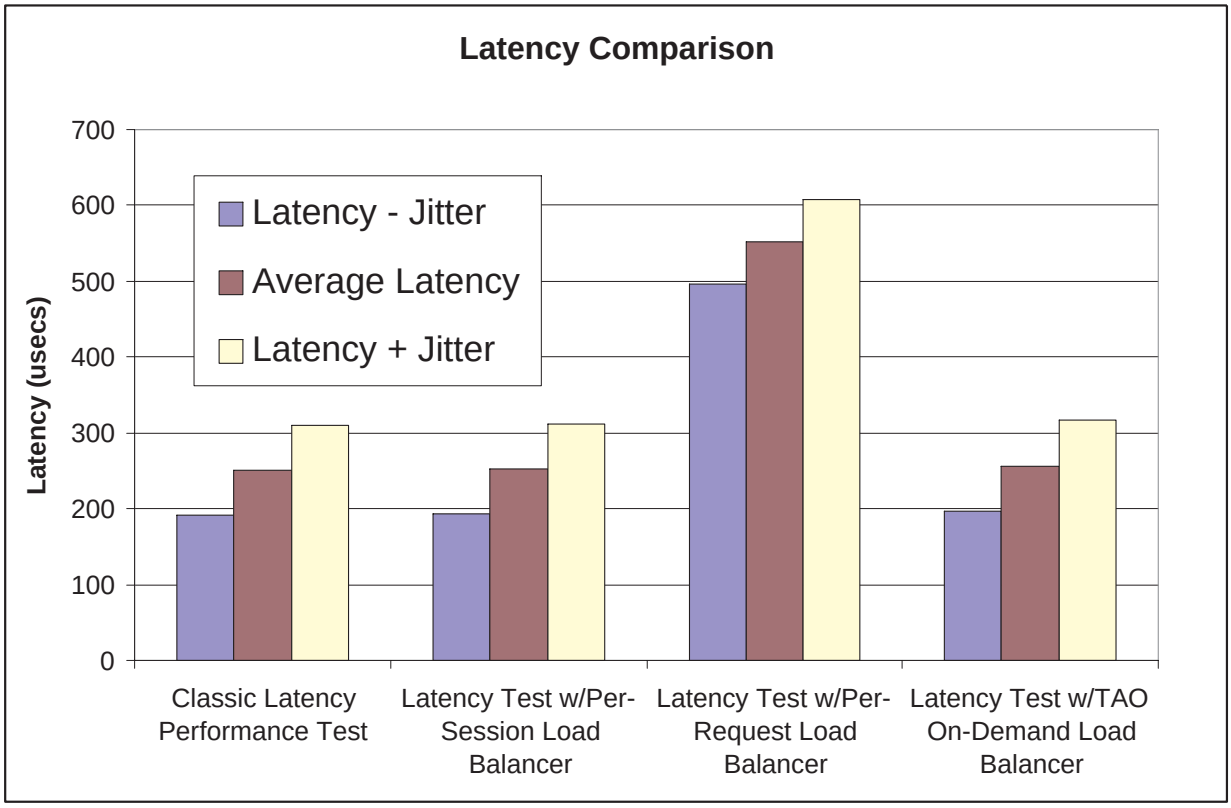

Figure 5.2: Load Balancing Latency Overhead

server is essentially a lightweight load balancer, this benchmark provides a baseline for the best performance achievable by a load balancer using a per-request client binding granularity.

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

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

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

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

Figure **??** shows how the average throughput differs between each load balancing strategy. Again, only one client and one server were used for this experiment. Not surprisingly, the throughput remained

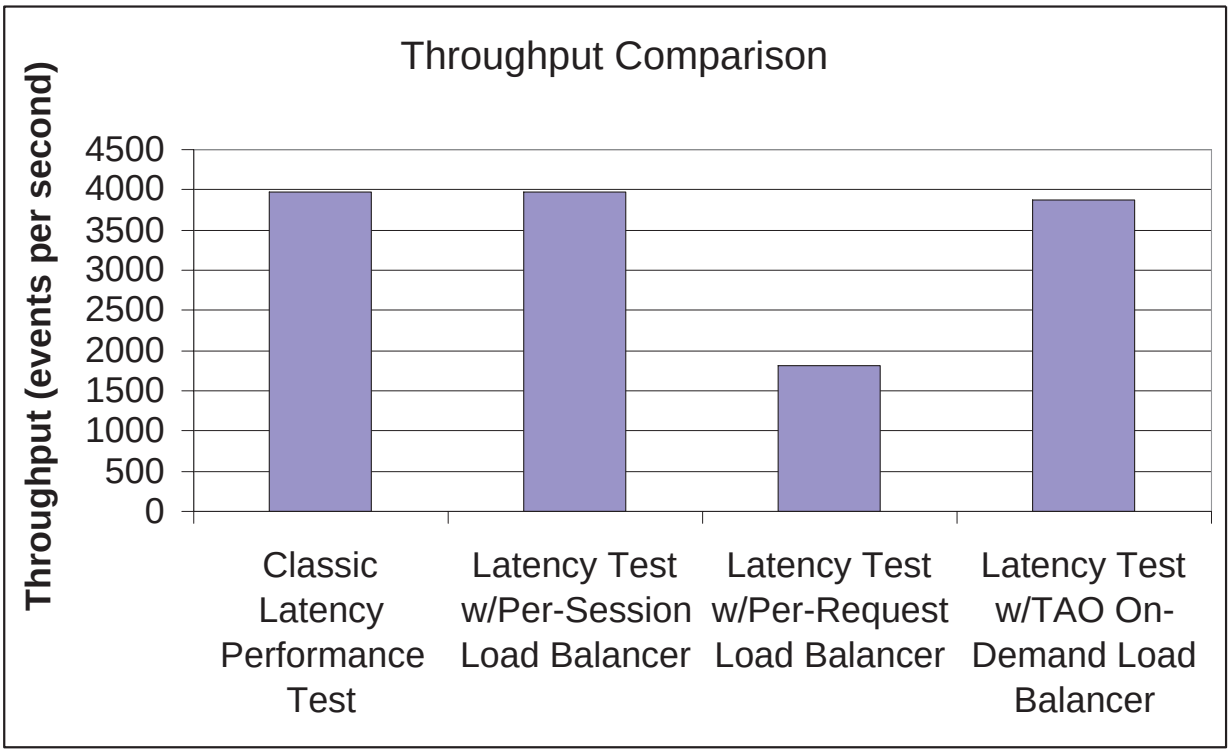

Figure 5.3: Load Balancing Throughput Overhead

basically unchanged for the non-adaptive per-session approach since only one out of 200,000 requests was forwarded. The remaining requests were all sent to directly to the server, *i.e.*, all requests were running at their maximum speed.

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

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

5.3 Load Balancing Strategy Effectiveness

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

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

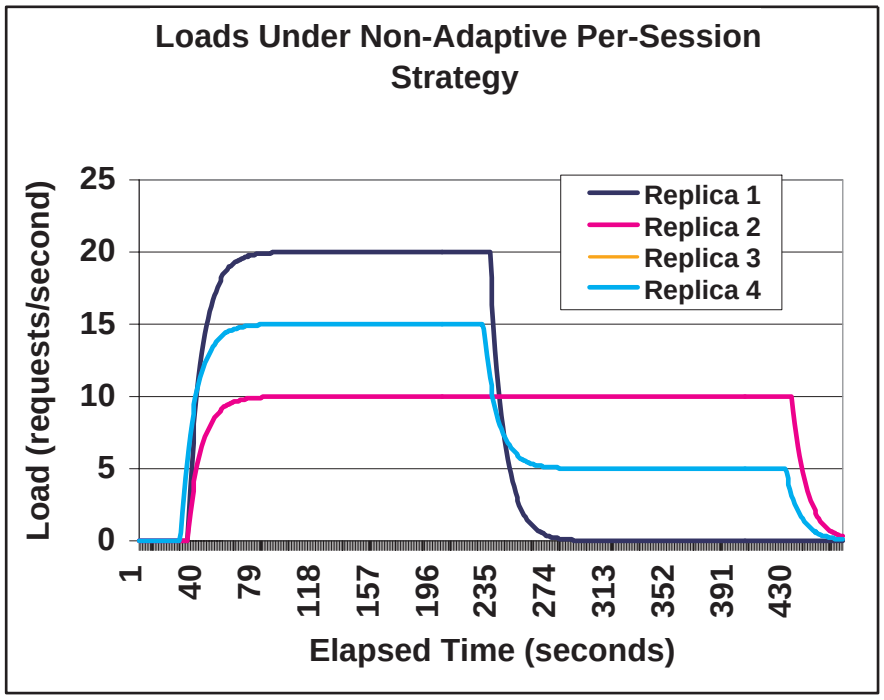

Figure 5.4: Effectiveness of Non-Adaptive Per-Session Load Balancing

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

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

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

Figure **??** illustrates the loads incurred on each of the `Latency` server replicas using non-adaptive per-session load balancing. The results quantify the degree to which loads across replicas become unbalanced by using this strategy. Since there is no feedback loop between the replicas and the load balancer, it is not possible to shift load from highly loaded replicas to less heavily loaded replicas.

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

• **TAO's adaptive load balancing strategy effectiveness:** This test demonstrated the benefits of an adaptive load balancing strategy. Therefore, we increased the load imposed by each client and increased the

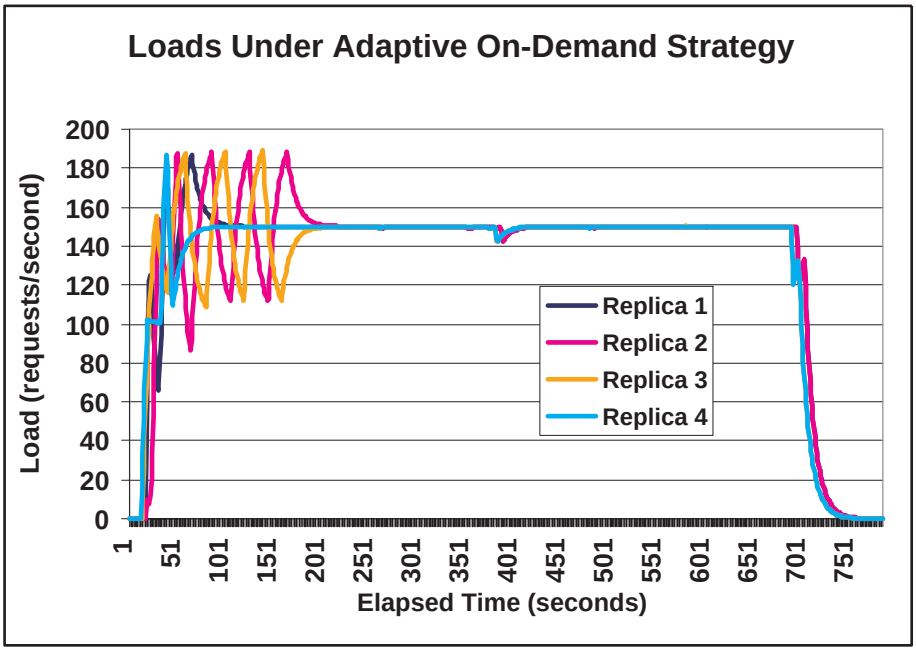

Figure 5.5: Effectiveness of Adaptive On-Demand Load Balancing

number of iterations from 200,000 to 750,000. Four clients running at 100 Hz and another four running at 50 Hz were started and ended simultaneously.

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

As Figure **??** illustrates, the loads across all four replicas fluctuated for a short period of time until an equilibrium load of 150 Hz was reached.[2] The initial load fluctuations result from the load balancer periodically rebinding clients to less loaded replicas. By the time a given rebind completed, the replica load had become imbalanced, at which point the client was rebound to another replica. These initial fluctuations are typical of the adaptive load balancing hazards discussed in Section **??**.

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

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

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

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

The actual time required to reach the equilibrium load depends greatly on the load balancing strategy. The example above was based on the minimum dispersion strategy described in Section **??**. A more sophisticated adaptive load balancing strategy could have been employed to improve the time to reach equilibrium. Regardless of the complexity of the adaptive load balancing strategy, these results show that adaptive load balancing strategies can maintain balanced loads across a given set of replicas.

5.4 Summary

Chapter 6

Related Work

This section compares and contrasts our work on load balancing with representative related work. This thesis concentrates primarily on architectural and optimization challenges associated with developing a scalable CORBA load balancing service.

6.1 Load Balancing at Various System Levels

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

Network-based load balancing. Network-based load balancing services make decisions based on the frequency at which a given site receives requests [**?**]. For example, routers [**?**] and DNS servers often perform network-based load balancing. Load balancing performed at the network level has the disadvantage that load balancing decisions are based solely on the destination of the request. The content of the request is often ignored. This form of load balancing also makes it difficult to select the load metric to be used when making balancing decisions.

OS-based load balancing. Load balancing at the operating system level [**?**, **?**, **?**] has the advantage of performing the balancing at multiple levels. That balancing is essentially transparent to a distributed application. However, it suffers from many of the same problems that network-based load balancing suffers from, such as inflexible load metric selection and not being able to take advantage of request content. OS-based load balancing may also be too coarse-grained for some distributed applications where it is the objects residing within a server, rather than the server process itself, that must be load balanced.

Middleware-based load balancing. Middleware-based load balancing provides the most flexibility in terms of influencing how a load balancing service makes decisions, and in terms of applicability to different types of distributed applications [**?**, **?**]. Load balancing at this level provides for straightforward selection of load metrics, in addition to the ability to make load balancing decisions based on the content of a request. Some middleware-based implementations integrate load balancing functionality into the ORB middleware [**?**] itself, whereas others implement load balancing support at the service level. The latter is the approach taken by the TAO next-generation load balancing service upon which the content of this thesis is based.

6.2 CORBA-based Load Balancing

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

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

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

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

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

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

6.3 Summary

Chapter 7

Future Work

This chapter describes several advanced load balancing features that address the inability of many load balancing services to satisfy the demanding optimization and quality of service (QoS) requirements exhibited by complex distributed systems. Those features include the following:

1. Stateful member
2. Decentralized load balancing
3. Fault tolerant load balancing
4. Improved load balancing algorithms

Furthermore, other related topics that should be explored are:

1. Middlware service composition
2. Starvation by competing object groups
3. Incompatible load balancing strategies
4. Relationships between CORBA Load Balancing and Data Parallel CORBA

Each of these topics are explored below.

7.1 Stateful Members

Context A server in a distributed application retains state that is used when servicing subsequent client requests, *e.g.*, the state can influence the results of future client requests.

Problem To enhance genericity and reuse, a load balancing service should be able to balance loads across stateful members. Thus, a load balancing service must ensure that state held by each member is consistent. In heterogeneous environments (*e.g.*, platforms with different binary formats) it is non-trivial to manage distributed state. A load balancing service must have *a priori* knowledge of the state contents to send or transfer state to other members. For example, the underlying middleware that actually handles state transfer must know the types of data in the member's state to marshal it correctly and efficiently for transport over a heterogeneous communication medium, such as the Internet. These requirements make it hard to fully automate load balancing of stateful members.

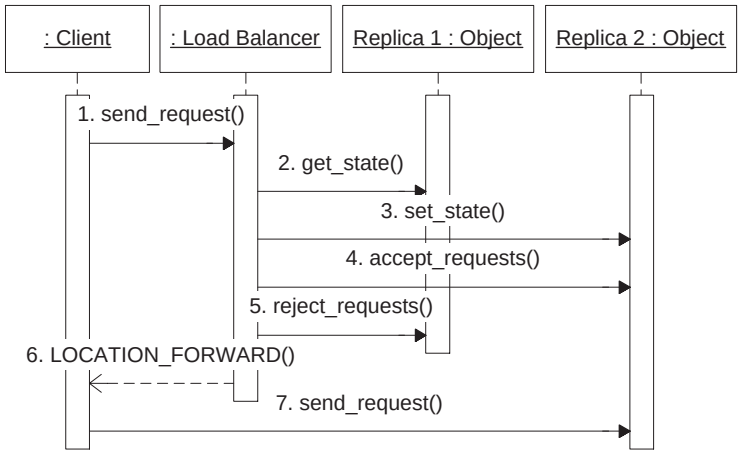

Figure 7.1: Load Balancing Stateful Members

Solution → the Memento Pattern To load balance members that retain state, some means of maintaining state consistency between members is necessary. The Memento pattern [**?**] can help to address this need by capturing internal state so that it may be restored at a later time. For example, a load balancing service could invoke `get_state` and `set_state` operations on a pair of members to transfer state between the members. These two methods are specified by the Memento pattern, and the member itself must implement them. Figure **??** illustrates the sequence of operations that occur when forwarding a client to a less load stateful member. These operations are outlined below:

1. A client makes a request. The request is intercepted by the load balancer transparently.
2. To transfer load to a new member, the load balancer obtains the state from the overloaded member using the `get_state` operation.
3. That state is then restored into the new underloaded member by invoking `set_state` operation on that underloaded member.
4. After the state transfer occurs, the new member can service client requests, and the load balancer notifies it that it can begin accepting requests.
5. The overloaded member must shed some of its load, so the load balancer notifies it that it should reject requests. This entails making the overloaded member redirect client requests back to the load balancer.
6. The load balancer now redirects the client to the new underloaded member transparently by means of the GIOP LOCATION_FORWARD message.
7. The client ORB reissues the request to the new less loaded member.

An alternative solution is provided by the CORBA Persistent State Service [**?**]. It extends the CORBA IDL[1] so that it becomes possible for distributed application developers to define precisely what the internal state is, *i.e.*, its format or schema. This *a priori* knowledge facilitates persistence at compile-time, and thus simplifies the automation and transfer of state in distributed applications.

[1] CORBA IDL is an *interface definition language* that is used to define interfaces supplied by servers to clients in distributed systems.

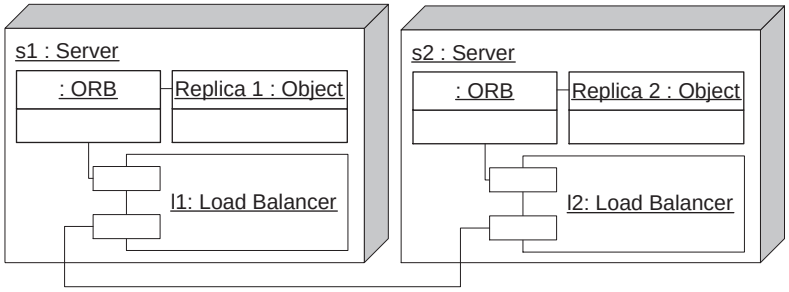

Figure 7.2: Federated Load Balancing

Both approaches require some modification to distributed applications. Thus, achieving truly transparent server-side load balancing of stateful members at the middleware level is non-trivial. Moreover, due to the required state transfers, load balancing stateful members incurs more overhead than stateless members. Although reliable multicast can be used to optimize state transfers, more network utilization is typically incurred by load balancing stateful members. As is often the case, application developers must settle for a trade-off between performance and quality of service.

7.2 Decentralized Load Balancing

Context A large group of distributed members is being load balanced. In addition to control requests sent from the load balancing service to the members, load information is sent to the load balancing service from each of these members.

Problem Adaptive middleware load balancing services are often *centralized, i.e.,* a single load balancing server manages client requests and member loads. Specifically, *one* load balancer performs all load balancing tasks for each distributed application. Although centralized load balancing services are simpler to design and implement, their centralization introduces a single point of failure, which can impede system reliability and scalability.

Solution → Federated Architecture To overcome the problems in centralized load balancing services, a *federated* load balancing architecture can be used to implement a more scalable and reliable load balancing service. In this model, load balancing is performed via a distributed, *i.e.,* decentralized, set of load balancers that collectively form a single *logical* load balancing service. This architecture is illustrated in Figure **??**.

The advantages of this architecture is that (1) a single point of failure does not exist and (2) no single bottleneck point exists either. Load balancing decisions are made cooperatively, *i.e.,* each load balancer can communicate with other balancers to decide how best to balance loads across a given group of members. Communication could, for example, be performed using reliable multicast to efficiently convey load information to other load balancers.

Decentralized load balancing schemes, such as the federated load balancing model described above, can potentially reduce the number of messages related to load balancing. In particular, a decentralized *hierarchical* architecture can be used to coalesce load balancing related messages as load reports are propagated up the hierarchy of load balancers. The reduced number of messages lowers network resource utilization, which in turn can improve overall performance of the distributed application being load balanced.

Applying the Solution in TAO The techniques described in Section **??** to ensure server-side load balancing transparency can be used to implement a federated load balancing service. In particular, the “distributed”

component of a federated load balancing service can reside in an interceptor that is installed transparently.

TAO's next-generation adaptive load balancer uses the Component Configurator pattern to dynamically load a factory object. That factory object creates an ORB initializer [**?**] object that registers the load balancing interceptor with an ORB each time an ORB is created. This design allows a portion of a federated load balancing service to transparently reside at multiple locations, *i.e.*, where ever members reside.

7.3 Fault Tolerant Load Balancing

Context A distributed application has *high availability* requirements. It must always be available to clients, *i.e.*, it must be *fault tolerant*.

Problem Centralized load balancing services are a single point of failure. For example, if a centralized load balancing service fails clients may not be able to have their requests serviced. Decentralized load balancing services potentially handle faults with greater ease than centralized ones. They are also distributed applications, however, and thus are susceptible to the same types of failures as the members they are load balancing.

Solution → a Fault Tolerance Service Since CORBA-based load balancing services are themselves CORBA applications, the standard CORBA Fault Tolerance service [**?**] can be used to provide the means by which a load balancing service remains highly available. Making a load balancing service fault tolerant by means of Fault Tolerant CORBA can alleviate one of the inherent problems with centralized load balancing: its single point of failure. It can also ensure that state within members is consistent, in the case of stateful members. This capability can simplify a load balancer implementation since the load balancer can delegate the task of ensuring state consistency between members to the Fault Tolerance service. One implementation of the CORBA Fault Tolerance service is DOORS [**?**, **?**]. Since DOORS itself is a CORBA service implemented using TAO, integrating it with TAO's load balancer should be straightforward, for example.

7.4 Improved Load Balancing Algorithms

Context The load conditions in a distributed application will change drastically at some point during the day. In some cases, the types of loads may also change. The times of day when these changes occur may not be knowable *a priori*. Moreover, the number of members servicing requests may also vary.

Problem Many load balancers only support a few load balancing algorithms. These load balancing algorithms may not be adequate at all times during the lifetime of a distributed application. If the client traffic changes substantially at run-time, however, loads across members will not be balanced effectively. Other related problems include situations where (1) several new members may be added to an object group dynamically, which cannot be predicted by a load balancer and (2) a poorly designed load balancing strategy cannot handle degenerate load balancing conditions, such as unstable members loads.

Solution → implement custom load balancing strategies Load balancing algorithms employed by the load balancing service can be implemented via the Strategy pattern [**?**]. This pattern allows a load balancing service to cope with degenerate load conditions, in addition to further generalizing the applicability of the load balancing service to other types of distributed applications. Figure **??** illustrates how this solution is deployed.

For example, load balancing algorithms/strategies that perform the following can be configured into a running load balancing service:

- Take into account past load trends when predicting future load conditions.

- Take advantage of sophisticated algorithms [**?**] that are designed specifically to restore system equilibrium when it is perturbed by external forces. In the case of load balancing, external forces could be additional client requests or transient loads generated by other applications running over the network and end-systems.

- Make load balancing decisions based on multiple load metrics, which requires the ability to send multiple loads in a single load report. For example, a load balancing strategy could receive a sequence of load metrics that correspond to multiple load readings, each of a different type, at a given location. The CORBA IDL for such a sequence of loads could be the following[2]:

```
module LoadBalancer {
  typedef unsigned long LoadId;
  struct Load {
    LoadId identifier;
    float value;
  };

  typedef sequence<Load> LoadList;
};
```

These approaches can improve the stability of adaptive load balancing strategies so that they perform better under heavy loads or under loads that change rapidly.

7.5 Middlware Service Composition

Determining how Quality-of-Service (QoS) requirements may be transparently fulfilled by composing middleware services, rather than by implementing ad hoc and proprietary services is important for distributed applications with stringent QoS requirements. Composition is used to reduce the amount of reproduced work. Taking the middleware approach instead of the proprietary approach prevents QoS-enabled application developers from reinventing the wheel.

However, naive composition of middleware services will not always provide the necessary QoS. For example, composing a load balancing service and a real-time middleware framework may not result in a scalable real-time application since one service may negate the effects of the other. Thus, simultaneously addressing scalability and real-time concerns, in addition to other QoS requirements, is non-trivial.

7.6 Starvation By Competing Object Groups

Adaptive load balancing behavior in the presence of multiple object groups of largely different resource requirements, each with members residing at the same locations is potentially complex. For example:

Given object groups `A`, `B` and `C`, and members `1...n` in each group, Table **??** illustrates a the locations of a number of members from these object groups.

If objects in group `B`, for example, incur much higher loads than groups `A` and `C`, group `B` may end up starving members from groups `A` and `C` at all locations. This means that invocations intended for members in group `A` and `C` will either just keep bouncing from location to location or the invocation will not go through at all. In such a case, the load balancer must have a priori knowledge of the loads that

[2]The forthcoming OMG Load Balancing and Monitoring specification already includes this IDL. TAO's OMG load balancing service implementation also supports this functionality. Custom load balancing strategy implementors need only to take advantage of it.

Location		
doc	rumba	sirion
Member `A.1`	Member `A.2`	Member `A.3`
Member `B.3`	Member `B.2`	Member `B.7`
Member `C.1`	Member `C.3`	Member `C.2`

Table 7.1: Competing Object Groups

members from a given object group will incur, or the load balancing strategy in effect must be able detect such starvation conditions and react accordingly.

7.7 Incompatible Load Balancing Strategies

With the current OMG load balancing architecture, each object group can be configured to use a different load balancing strategy. TAO's load balancer, for example, allows this. Such flexibility is nice, but it may turn out that some load balancing strategies may negate the effects of others when members from different object groups exist at the same location, such as in the scenario shown in Table **??**. For example, group `A`'s configured strategy may try to be fair about resource sharing while group `B`'s configured strategy may be greedy. This can cause some strange interaction loops in the load manager's `LoadAnalyzer` component since one strategy may detect an overload condition while the other believes that there is no overload condition.

7.8 Relationships between CORBA Load Balancing and Data Parallel

7.9 Summary

The load balancing research detailed in thesis produced results that are very useful in "real world" distributed applications with high scalability requirements. However, there still remains a large number of load balancing issues that should be explored before a load balancing service implementation can be considered robust enough for most, if not all, distributed applications. Such issues include improved reduction in the load balancer's own resource utilization, composability with other services, inter-object group interaction stability, and compatibility of load balancing strategies employed by object groups competing for the same resources.

Chapter 8

Conclusion

As network-centric computing becomes 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, load balancing mechanisms can be used to distribute system load across object group members residing on multiple servers.

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

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

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

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

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

TAO's load balancing service implementation is based entirely on standard features in CORBA, 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. We believe that further research and documentation of the effective architectures and design patterns used in the implementation of higher-level CORBA services is required to advance the state of the practice and to allow application developers to make better decisions when designing their systems.

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

As distributed applications become increasingly complex, broader in scope, and more dynamic in their behavior, the ability of non-adaptive middleware load balancing services to improve overall performance decreases. In general, the utility of non-adaptive middleware load balancing services decreases because they are (1) designed for a specific application and (2) because they cannot adapt to changing run-time load conditions. Moreover, many load balancing services that do adapt to changing load conditions cannot handle a large number of operating/load conditions or require modifications to distributed applications.

To optimize overall performance, scalability, and reliability, middleware-based load balancing services should provide the functionality detailed in this paper:

- Server-side transparency
- State migration
- Differents load monitoring granularity levels
- Federated load balancing architectures
- Fault tolerance
- Extensible load balancing strategy support
- Run-time control of group member life times

We believe that these features are essential to implement a generalized, highly effective and optimized adaptive CORBA load balancing service.

TAO's next-generation load balancer will support the functionality outlined above. Transparent server-side load balancing for stateless members will be supported by the standard CORBA portable interceptors [**?**] mechanism. Federated load balancing will be implemented via reliable multicast. State migration will be supported by using the CORBA Persistent State Service [**?**] being developed for TAO. Different load monitoring granularity levels will be supported via the CORBA portable interceptor mechanism, in addition to hierarchical load monitoring. Basic fault tolerance will be supported through CORBA Fault Tolerance service implementation [**?**, **?**] currently being developed for TAO. Extensible load balancing strategies are already supported by TAO. Finally, TAO's run-time control of group member life times will capitalize on the interface provided by the CORBA Fault Tolerance specification.

Bibliography

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

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

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

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

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

[6] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education Limited, Harlow, England, 2001.

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

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

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

[10] Vittorio Ghini, Fabio Panzieri, and Marco Roccetti. Client-centered Load Distribution: A Mechanism for Constructing Responsive Web Services. In *Proceedings of the 34th Hawaii International Conference on System Sciences - 2001*, Hawaii, USA, 2001.

[11] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, 1999.

[12] Key Shiu Ho and Hong Va Leong. An Extended CORBA Event Service with Support for Load Balancing and Fault-Tolerance. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, Antwerp, Belgium, September 2000. OMG.

[13] Chi-Chung Hui and Samuel T. Chanson. Improved Strategies for Dynamic Load Balancing. *IEEE Concurrency*, 7(3), July 1999.

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

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

[16] Ervin Johnson and ArrowPoint Communications. A Comparative Analysis of Web Switching Architectures. www.arrowpoint.com/solutions/white_papers/ws_archv6.html, 1998.

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

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

[19] Markus Lindermeier. Load Management for Distributed Object-Oriented Environments. In *Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, September 2000. OMG.

[20] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik. Applying Patterns to Improve the Performance of Fault-Tolerant CORBA. In *Proceedings of the 7^{th} International Conference on High Performance Computing (HiPC 2000)*, Bangalore, India, December 2000. ACM/IEEE.

[21] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik. DOORS: Towards High-performance Fault-Tolerant CORBA. In *Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, September 2000. OMG.

[22] Object Management Group. *Fault Tolerant CORBA Specification*, OMG Document orbos/99-12-08 edition, December 1999.

[23] Object Management Group. *Persistent State Service 2.0 Specification*, OMG Document orbos/99-07-07 edition, July 1999.

[24] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.3 edition, June 1999.

[25] Object Management Group. *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 edition, April 2000.

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

[27] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. An Efficient Adaptive Load Balancing Service for CORBA. *IEEE Distributed Systems Online*, 2(3), March 2001.

[28] Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt. The Design of an Adaptive CORBA Load Balancing Service. *IEEE Distributed Systems Online*, 2(4), April 2001.

[29] Nat Pryce. Abstract Session. In Brian Foote, Neil Harrison, and Hans Rohnert, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, Massachusetts, 1999.

[30] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

[31] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *Proceedings, IEEE Aerospace*. IEEE, 1997.

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

[33] Douglas C. Schmidt, Vishal Kachroo, Yamuna Krishnamurthy, and Fred Kuhns. Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications. *IEEE Communications Magazine*, 38(10):112–123, October 2000.

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

[35] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

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

[37] Nanbor Wang, Kirthika Parameswaran, and Douglas C. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware. In *Proceedings of the* 6^{th} *Conference on Object-Oriented Technologies and Systems*, pages 103–118, San Antonio, TX, Jan/Feb 2000. USENIX.

[38] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman. DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable Real-Time Systems. In *IFACs 15th Symposium on Distributed Computer Control Systems (DCCS98)*. IFAC, 1998.

Part II

Appendices

Appendix A

Load Balancing at Different Layers

Load balancing mechanisms can be provided in any or all of the following layers in a distributed system:

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

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

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

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

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

Middleware-based load balancing can be used in conjunction with the specialized network-based and OS-based load balancing mechanisms outlined above. It can also be applied on top of commodity-off-the-shelf (COTS) networks and operating systems, which helps reduce cost. In addition, middleware-based load balancing can provide semantically-rich customization hooks to perform load balancing based

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

on a wide range of application-specific load balancing conditions, such as run-time I/O vs. CPU overhead conditions.

- **Application-based load balancing:**

Appendix B

CORBA Load Balancing and Monitoring Interfaces and Types

The forthcoming CORBA Load Balancing and Monitoring specification defines several CORBA IDL modules, interfaces and types. This appendix lists the full IDL available in that specification.

B.1 CORBA Load Balancing and Monitoring IDL

The IDL defined in the CORBA Load Balancing and Monitoring specification is divided into two core modules, `PortableGroup` and `CosLoadBalancing`. Listings of each follow.

B.1.1 `PortableGroup` IDL Module

The `PortableGroup` IDL module defines a set of interfaces and types useful for object group creation, object group management and object group property management. It is based on the some of the original interfaces and types defined in the CORBA Fault Tolerance specification. Interfaces and types not specific to fault tolerance that are applicable to other technologies that require group management, such as load lalancing and parallel computation, have been factored out into the `PortableGroup` IDL module. Other CORBA technologies, such as Data Parallel CORBA, Unreliable Multicast and Load Balancing and Monitoring have adopted this core group management IDL. The `PortableGroup` module will be standardized through the CORBA Load Balancing and Monitoring specification.

```
#ifndef _PORTABLEGROUP_IDL_
#define _PORTABLEGROUP_IDL_

#include <CosNaming.idl>
#include <IOP.idl>
#include <GIOP.idl>
#include <orb.idl>

#pragma prefix "omg.org"

module PortableGroup {

  // Specification for Interoperable Object Group References
  typedef string GroupDomainId;
  typedef unsigned long long ObjectGroupId;
  typedef unsigned long ObjectGroupRefVersion;
```

```
struct TagGroupTaggedComponent { // tag = TAG_GROUP;
  GIOP::Version component_version;
  GroupDomainId group_domain_id;
  ObjectGroupId object_group_id;
  ObjectGroupRefVersion object_group_ref_version;
};
typedef sequence <octet> GroupIIOPProfile; // tag = TAG_GROUP_IIOP

// Specification of Common Types and Exceptions for Group Management
interface GenericFactory;

typedef CORBA::RepositoryId _TypeId;
typedef Object ObjectGroup;
typedef CosNaming::Name Name;
typedef any Value;

struct Property {
  Name nam;
  Value val;
};

typedef sequence<Property> Properties;
typedef Name Location;
typedef sequence<Location> Locations;
typedef Properties Criteria;

struct FactoryInfo {
  GenericFactory the_factory;
  Location the_location;
  Criteria the_criteria;
};
typedef sequence<FactoryInfo> FactoryInfos;

typedef long MembershipStyleValue;
const MembershipStyleValue MEMB_APP_CTRL = 0;
const MembershipStyleValue MEMB_INF_CTRL = 1;

typedef FactoryInfos FactoriesValue;
typedef unsigned short InitialNumberMembersValue;
typedef unsigned short MinimumNumberMembersValue;

exception InterfaceNotFound {};
exception ObjectGroupNotFound {};
exception MemberNotFound {};
exception ObjectNotFound {};
exception MemberAlreadyPresent {};
exception ObjectNotCreated {};
exception ObjectNotAdded {};
exception UnsupportedProperty {
```

```
  Name nam;
  Value val;
};

exception InvalidProperty {
  Name nam;
  Value val;
};

exception NoFactory {
  Location the_location;
  _TypeId type_id;
};

exception InvalidCriteria {
  Criteria invalid_criteria;
};

exception CannotMeetCriteria {
  Criteria unmet_criteria;
};

// Specification of PropertyManager Interface
interface PropertyManager {

  void set_default_properties (in Properties props)
    raises (InvalidProperty, UnsupportedProperty);

  Properties get_default_properties();

  void remove_default_properties (in Properties props)
    raises (InvalidProperty, UnsupportedProperty);

  void set_type_properties (in _TypeId type_id, in Properties overrides)
    raises (InvalidProperty, UnsupportedProperty);

  Properties get_type_properties(in _TypeId type_id);

  void remove_type_properties (in _TypeId type_id, in Properties props)
    raises (InvalidProperty, UnsupportedProperty);

  void set_properties_dynamically
    (in ObjectGroup object_group, in Properties overrides)
    raises (ObjectGroupNotFound,
            InvalidProperty,
            UnsupportedProperty);

  Properties get_properties (in ObjectGroup object_group)
    raises (ObjectGroupNotFound);
}; // endPropertyManager
```

```
// Specification of ObjectGroupManager Interface
interface ObjectGroupManager {
  ObjectGroup create_member (in ObjectGroup object_group,
                             in Location the_location,
                             in _TypeId type_id,
                             in Criteria the_criteria)
    raises (ObjectGroupNotFound,
            MemberAlreadyPresent,
            NoFactory,
            ObjectNotCreated,
            InvalidCriteria,
            CannotMeetCriteria);

  ObjectGroup add_member (in ObjectGroup object_group,
                          in Location the_location,
                          in Object member)
    raises (ObjectGroupNotFound,
            MemberAlreadyPresent,
            ObjectNotAdded);

  ObjectGroup remove_member (in ObjectGroup object_group,
                             in Location the_location)
    raises (ObjectGroupNotFound, MemberNotFound);

  Locations locations_of_members (in ObjectGroup object_group)
    raises (ObjectGroupNotFound);

  ObjectGroupId get_object_group_id (in ObjectGroup object_group)
    raises (ObjectGroupNotFound);

  ObjectGroup get_object_group_ref (in ObjectGroup object_group)
    raises (ObjectGroupNotFound);

  Object get_member_ref (in ObjectGroup object_group,
                         in Location loc)
    raises (ObjectGroupNotFound, MemberNotFound);

}; // end ObjectGroupManager


// Specification of GenericFactory Interface
interface GenericFactory {
  typedef any FactoryCreationId;

  Object create_object (in _TypeId type_id,
                        in Criteria the_criteria,
                        out FactoryCreationId factory_creation_id)
    raises (NoFactory,
```

```
                ObjectNotCreated,
                InvalidCriteria,
                InvalidProperty,
                CannotMeetCriteria);

    void delete_object (in FactoryCreationId factory_creation_id)
      raises (ObjectNotFound);

  }; // end GenericFactory

}; // end PortableGroup

#endif  /* _PORTABLEGROUP_IDL_ */
```

B.1.2 `CosLoadBalancing` IDL Module

The `CosLoadBalancing` IDL module contains all the interfaces and types necessary to facilitate load balancing and monitoring. Where possible, it leverages the existing interfaces and types in the `PortableGroup` IDL module.

```
#ifndef COSLOADBALANCING_IDL
#define COSLOADBALANCING_IDL

#include <PortableGroup.idl>
#include <orb.idl>

#pragma prefix "omg.org"

module CosLoadBalancing
{
  const IOP::ServiceId LOAD_MANAGED = 123456;  // @todo TBA by OMG

  typedef PortableGroup::Location Location;
  typedef PortableGroup::Properties Properties;

  // Types used for obtaining and/or reporting loads
  typedef unsigned long LoadId;

  // OMG defined LoadId constants.
  const LoadId CPU = 0;
  const LoadId Disk = 1;
  const LoadId Memory = 2;
  const LoadId Network = 3;

  struct Load {
    LoadId id;
    float value;
  };
  typedef sequence<Load> LoadList;
```

```
exception MonitorAlreadyPresent {};
exception LocationNotFound {};
exception LoadAlertNotFound {};
exception LoadAlertAlreadyPresent {};
exception LoadAlertNotAdded {};

exception StrategyNotAdaptive {};

interface LoadManager;

interface Strategy
{
  readonly attribute string name;

  Properties get_properties ();

  // Report loads at given location to the LoadManager.
  void push_loads (in PortableGroup::Location the_location,
                   in LoadList loads)
    raises (StrategyNotAdaptive);

  // Get loads, if any, at the given location.  Load balancing
  // strategies may use this method to query loads at specific
  // locations.  Returned loads are the effective loads computed by
  // the Strategy, as opposed to the raw loads maintained by the
  // LoadManager.
  LoadList get_loads (in LoadManager load_manager,
                      in PortableGroup::Location the_location)
    raises (LocationNotFound);

  // Return the next member from the given object group which will
  // requests will be forward to.
  Object next_member (in PortableGroup::ObjectGroup object_group,
                      in LoadManager load_manager)
    raises (PortableGroup::ObjectGroupNotFound,
            PortableGroup::MemberNotFound);

  // Ask the Strategy to analyze loads, and enable or disable alerts
  // at object group members residing at appropriate locations.
  // oneway void analyze_loads (in LoadManager load_manager);

  // The given loads at the given location should no longer be
  // considered when performing load analysis.
  void location_removed (in PortableGroup::Location the_location)
    raises (LocationNotFound);
};

interface CustomStrategy : Strategy {
};
```

```
// Property value for built-in load balancing Strategy.
struct StrategyInfo
{
  string name;
  Properties props;
};

interface LoadAlert
{
  // Forward requests back to the load manager via the object group
  // reference.
  void enable_alert ();

  // Stop forwarding requests, and begin accepting them again.
  void disable_alert ();
};

// Interface that all load monitors must implement.
interface LoadMonitor
{
  // Retrieve the location at which the LoadMonitor resides.
  readonly attribute Location the_location;

  // Retrieve the current load at the location LoadMonitor resides.
  readonly attribute LoadList loads;
};

// Specification of LoadManager Interface
interface LoadManager
  : PortableGroup::PropertyManager,
    PortableGroup::ObjectGroupManager,
    PortableGroup::GenericFactory
{
  // For the PUSH load monitoring style.
  void push_loads (in PortableGroup::Location the_location,
                   in LoadList loads);

  // Return the raw loads at the given location, as opposed to the
  // potentially different effective loads returned by the
  // Strategy::get_loads() method.
  LoadList get_loads (in PortableGroup::Location the_location)
    raises (LocationNotFound);

  // Inform member at given location of load alert condition.
  void enable_alert (in PortableGroup::Location the_location)
    raises (LoadAlertNotFound);

  // Inform member at given location that load alert condition has
  // passed.
  void disable_alert (in PortableGroup::Location the_location)
```

```
      raises (LoadAlertNotFound);

    // Register a LoadAlert object for the member at the given
    // location.
    void register_load_alert (in PortableGroup::Location the_location,
                              in LoadAlert load_alert)
      raises (LoadAlertAlreadyPresent,
              LoadAlertNotAdded);

    // Retrieve the LoadAlert object for the member at the given
    // location.
    LoadAlert get_load_alert (in PortableGroup::Location the_location)
      raises (LoadAlertNotFound);

    // Remove (de-register) the LoadAlert object for the member at the
    // given location.
    void remove_load_alert (in PortableGroup::Location the_location)
      raises (LoadAlertNotFound);

    // The following load monitor methods are only used for the PULL
    // load monitoring style.

    // Register a LoadMonitor object for the given location.
    void register_load_monitor (in LoadMonitor load_monitor,
                                in PortableGroup::Location the_location)
      raises (MonitorAlreadyPresent);

    // Retrieve the LoadMonitor object for the given location.
    LoadMonitor get_load_monitor (in PortableGroup::Location the_location)
      raises (LocationNotFound);

    // Remove (de-register) the LoadMonitor object for the given
    // location.
    void remove_load_monitor (in PortableGroup::Location the_location)
      raises (LocationNotFound);

  };

};

#endif  /* COSLOADBALANCING_IDL */
```