The Design and Performance of an Adaptive CORBA Load Balancing Service

Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt
{ossama, coryan, schmidt}@uci.edu
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697-2625, USA *

November 14, 2000

Abstract

CORBA is increasingly popular as distributed object computing middleware for systems with stringent quality of service (QoS) requirements, including scalability and dependability. One way to improve the scalability and dependability of CORBA-based applications is to balance system processing load among multiple server hosts. Load balancing can help improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. Likewise, it can help improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures.

This paper presents four contributions to research on CORBA-based load balancing. First, we describe deficiencies with common load-balancing techniques, such as introducing unnecessary overhead or not adapting dynamically to changing load conditions. Second, we present a novel adaptive load balancing service that can be implemented efficiently using standard CORBA features. Third, we describe the key design challenges we faced when adding this load balancing service to our CORBA ORB (TAO) and outline how we resolved the challenges by applying patterns. Finally, we present the results of benchmark experiments that evaluate the pros and cons of different load balancing strategies empirically by measuring the overhead of each strategy and showing how well each strategy balances system load.

Keywords: Middleware, patterns, CORBA, load balancing.

1 Introduction

Motivation: The growth of online Internet services during the past decade has increased the demand for scalable and dependable distributed computing systems. For example, many e-commerce systems and online stock trading systems must service a large number of clients that transmit a high, often bursty, volume of requests. To protect initial hardware investments and avoid overcommitting resources it is desirable to scale up these systems incrementally. For example, servers can be connected via high-speed networks, with new servers purchased as the number of clients increase or extra server cycles leased during peak hours.

An increasingly popular and cost effective technique for improving the performance of these networked server architectures is *load balancing*, where hardware and/or software mechanisms determine which server will execute each client application request. Load balancing mechanisms distribute client workload equitably among various back-end servers to obtain the best response time possible given a particular load. These 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 the current load conditions. The client then contacts the designated back-end server, unaware that its next DNS resolution could select a different server. It is also possible to use routers that can bind a TCP flow to any of several back-end servers based on the current load conditions and then use that binding for the duration of the flow.

High volume Web sites often use network-based load balancing at the *network* layer (layer 3) and *transport* layer (layer 4). Layer 3 and 4 load balancing (referred to as "switching" in the trade literature [1]), use the IP address/hostname and port, respectively, to determine where to forward packets. Load balancing at these layers is somewhat limited, how-

*This work was funded in part by Automated Trading Desk, BBN, Cisco, DARPA contract 9701516, and Siemens MED.

ever, by the fact that they do not take into account the content of client requests. Instead, higher-layer mechanisms–such as the so-called layer 5 switching described below–perform load balancing in accordance with the content of requests, such as pathname information within a URL.

- **OS-based load balancing:** This type of load balancing is provided by distributed operating systems via *clustering*, *load sharing*, and *process migration* [2] mechanisms. Clustering is an economical means to achieve high-availability and high-performance by combining the individual processing power of many commodity computers to improve overall system processing power. Processes can be distributed transparently among computers in the cluster.

Clusters generally employ load sharing and process migration.[1] Balancing load across processors–or more generally across network nodes–can be achieved via process migration mechanisms, where the state of a given process is transferred from one node to another. Transferring the state of a process may require significant platform infrastructure support to handle potential architectural differences between nodes, and may limit its applicability to applications and 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 [1] 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 paper focuses on another type of middleware-based load balancing that can be performed using *object request brokers* (ORBs), such as CORBA [3]. ORBs allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [4]. 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 makes it an attractive choice to reduce costs. In addition, middleware-based load balancing can provide semantically-rich customization hooks to perform load balancing based on a wide range of application-specific load balancing conditions, such as run-time I/O vs. CPU overhead conditions.

[1] "Load sharing" should not be confused with "load balancing." For example, processing resources may be *shared* among processors but they need not necessarily be *balanced*.

Our previous research on middleware has examined many dimensions of ORB endsystem design, including static [5] and dynamic [6] scheduling, event processing [7], I/O subsystem [8] and pluggable protocol [9] integration, synchronous [10] and asynchronous [11] ORB Core architectures, ORB fault tolerance [12], systematic benchmarking of multiple ORBs [13], patterns for ORB extensibility [14] and ORB performance [15]. This paper focuses on another dimension in the CORBA research domain: *the design and performance of middleware-based load balancing mechanisms developed using standard CORBA*. Our approach is based on standard CORBA features available in any ORB compliant with the CORBA 2.3 [3] (or later) specification. This approach can also be generalized to other distributed object computing middleware, such as COM+ and Java RMI, that offer similar features.

CORBA load balancing example: To illustrate the benefits of middleware-enabled load balancing, consider the CORBA-based online stock trading system shown in Figure 1. A distributed online stock trading system creates sessions through which trading is conducted. This system consists of multiple back-end servers (replicas) that process session creation requests sent by clients over a network. In essence, these replicas are "session factories" [16]. Since the sole purpose of session factories is to create stock trading sessions, they need not retain state. Moreover, in this type of system client requests arrive dynamically, not deterministically, and the duration of each request many not be known *a priori*.

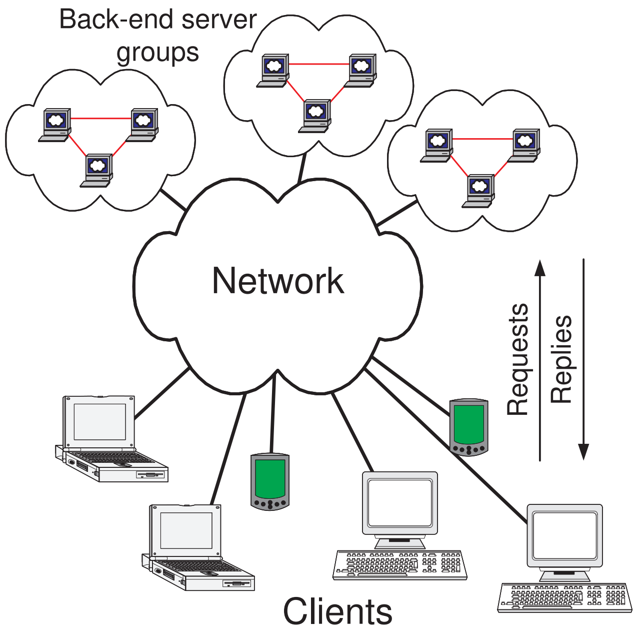

Figure 1: A Distributed Online Stock Trading System

CORBA is an effective choice for this online stock trading system due to the inherent distribution and 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 the complexities of distribution, thereby allowing them to concentrate their efforts on stock trading business logic.

In theory, having multiple back-end servers can (1) increase the scalability and dependability of the system, (2) reduce the initial investment when the number of clients is small, and (3) allow the system to scale up gracefully to handle more clients and processing workload in larger configurations. In practice, achieving this degree of scalability and dependability requires a sophisticated load balancing service. Ideally, integrating support for such a 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.

CORBA's rich set of features provides the means by which such a load balancing service can be realized. The CORBA load balancing service described in this paper is a prime candidate to fulfill the needs of applications that require increased scalability, such as the online stock trading system outlined above. In contrast, neither the network-based nor OS-based load balancing solutions provide as straightforward and portable a means of adapting load balancing decisions based on application-level request characteristics, such as content and duration.

Paper organization: The remainder of this paper is organized as follows: Section 2 summarizes the requirements of CORBA-based load balancing services and outlines the pros and cons of alternative solution architectures; Section 3 describes the design of our load balancing service, which is based on standard CORBA features and implemented using the TAO open-source[2] CORBA-compliant ORB; Section 4 evaluates the performance of alternative load balancing strategies empirically; Section 5 compares our adaptive middleware-based load balancing service with related work and outlines our future plans for enhancing TAO's load balancing service; and Section 6 presents lessons learned and concluding remarks. For completeness, Appendix A presents an overview of the standard CORBA reference architecture [3].

[2]The source code and documentation for TAO can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

2 Requirements and Alternative Solution Architectures

This section first describes the types of requirements that a CORBA-compliant load balancing service should be designed to address. Next, it presents an overview of several alternative load balancing architectures suitable for CORBA-based applications.

2.1 Requirements for a CORBA Load Balancing Service

The OMG CORBA specification provides the capabilities necessary to support load balancing. In particular, a CORBA load balancing service can take full advantage of the *request forwarding* mechanism mandated by the CORBA specification [3]. A CORBA server application can use this mechanism by raising a `PortableServer::ForwardRequest` exception; the server ORB catches this exception and then returns a LOCATION_FORWARD GIOP message. When the client ORB receives this message, the CORBA specification requires it to (1) re-issue the request to the new location specified in the LOCATION_FORWARD response and (2) to continue using that location until either the communication fails or the client is redirected again. Thus, the ORB can forward client requests to other servers *transparently*, *portably*, and *interoperably*.

The CORBA specification, however, does not *standardize* load balancing interfaces. Nor does it specify load balancing mechanisms, which are left as implementation decisions for ORB providers. Below, therefore, we describe the key requirements that CORBA load balancing services should be designed to address.

Support an object-oriented programming model: In the CORBA programming model clients invoke operations on objects. Thus, a CORBA load balancing service should be based on the concept of objects, rather than, *e.g.*, processes or TCP/IP addresses. Moreover, a load balancing service and ORB should coordinate the interactions amongst *multiple* object replicas.

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 the implementation of clients should be as simple as possible. A CORBA load balancing service that follows this principle should therefore require no changes to any of its clients.

Ideally, load balancing should require few modifications to servers, as well. This goal is hard to achieve in practice, however. 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 object's servant[3] 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 this 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 [17], they do not apply in complex Internet or military [18] 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 load distribution: A CORBA load balancing service must increase system scalability by maximizing 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.

Increase system dependability: Load balancing services can also handle certain types of server failures. For example, by using administrative interfaces or automated policies, clients that access a crashed or failing server can be migrated to other servers until the failure is resolved. Load balancing services need not provide full fault-tolerance capabilities, however, *i.e.*, it is not the role of a load balancing service to detect and mask failures [19, 20]. Instead, they should provide mechanisms to handle those failures efficiently when they are detected by administrators or other components in the system.

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 removal of replicas for upgrades, preemptive maintenance, or reassignment of system resources.

Minimal overhead: A CORBA load balancing service should not introduce undue latency or networking overhead since otherwise it may actually reduce–rather than enhance–overall system performance. In particular, an implementation that (1) increases the 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. Section 4 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.
- **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. Thus, clients can 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.

2.2 Overview of Alternative CORBA Load Balancing Strategies and Architectures

2.2.1 Load Balancing Strategies

There are various strategies for designing CORBA load balancing services. These strategies can be classified along the following orthogonal dimensions:

Client binding granularity: A load balancer *binds* a client request to a particular object 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

[3] The servant is an implementation language entity that provides the object functionality.

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*, 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 entails forcing a client to send its requests to a replica different from 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 an 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 in 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.

2.2.2 Load Balancing Architectures

By combining the strategies described above in various ways, it is possible to create the alternative load balancing architectures described below. In the ensuing discussion, we refer back to the requirements presented in Section 2.1 to evaluate the pros and cons of these strategies *qualitatively*. Section 4 then evaluates these different strategies *quantitatively*.

Non-adaptive per-session architectures: One way to design a CORBA load balancer is to select the target back-end server when a client/server session is first established, *i.e.*, when a client obtains an object reference to a service and connects to that service, as shown in Figure 2. 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 the load conditions on that server. This architecture is suitable for load balancing policies that implement round-robin or randomized balancing algorithms.

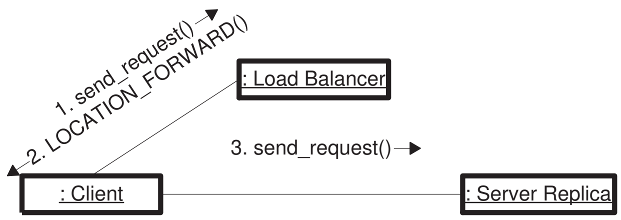

Figure 2: A Non-Adaptive Per-Session Architecture

Different clients can be directed to different object replicas by either using (1) a middleware activation daemon, such as a CORBA Implementation Repository [21] or (2) a lookup service, such as the CORBA Naming or Trading service. For example, Orbix [22] provides an extension to the CORBA Naming Service that returns references to object replicas in either a random or round-robin order.

Load balancing services based on a per-session client binding architecture can be implemented to support many requirements defined in Section 2.1. For example, per-session client binding architectures generally satisfy requirements for (1) application transparency, (2) increased system dependability, (3) minimal overhead, and (4) 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 suffer, therefore, 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 4.2.

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, clients could be re-bound at regular time intervals, for example.

Adaptive per-session architecture: This architecture is very similar to the non-adaptive per-session approach. Unlike it, however, it can use run time load information to select the replica, this information can be used to avoid binding new clients to already loaded replicas. However, this only represents and slight improvement, as clients can change the amount of load they generate after the binding decision is made. In that situation, the adaptive on-demand architecture offers a clear advantage as it can respond to dynamic changes in the load generated by each client.

Adaptive per-request architectures: A more adaptive request architecture for CORBA load balancing is shown in Figure 3. This design introduces a front-end server, which is a

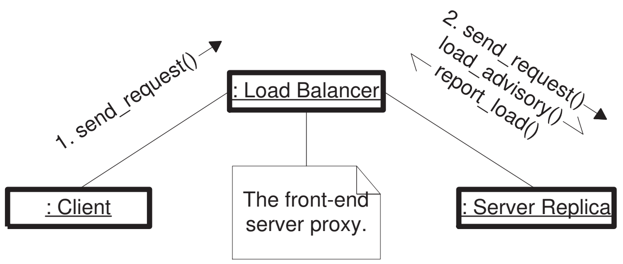

Figure 3: An Adaptive Per-request Architecture

proxy [16, 23] that receives all client requests. In this case, the "front-end server" is the load balancer. The intermediate server proxy selects an appropriate back-end server replica in accordance with its load balancing policy and forwards the request to the back-end server. The front-end server proxy waits for the back-end server replica's reply to arrive and then returns it to the client.

The primary benefit of the 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, this architecture can also introduce excessive latency and network overhead because each request is processed by the 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 to the front-end server.

In addition, to ensure that the system is scalable and dependable (*e.g.*, no single point of failure), multiple intermediate servers may be required. This configuration in turn requires complex algorithms that propagate the current load information to all front-end servers. It also requires a mechanism to assign clients to the correct front-end server. In a sense, therefore, the load balancing problem must be solved both for back-end *and* front-end servers, which complicates system design and implementation.

Adaptive on-demand architecture: This architecture is the focus of the paper. As shown in Figure 4, clients receive an

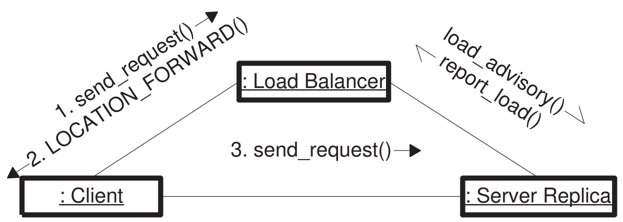

Figure 4: An Adaptive On-Demand Architecture

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 redirected again or 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 they use the standard CORBA LOCATION_FORWARD mechanism to redirect subsequent clients back to the load balancer. Upon receipt of a LOCATION_FORWARD message, a standard CORBA client ORB re-contacts the load balancer, which then redirects the client transparently to a less heavily 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. This strategy satisfies most of the requirements outlined in Section 2.1. 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 server replicas 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 load balancer control messages, which we call *load advisories*.

It is possible to overcome some drawbacks of adaptive on-demand load balancers, however, by applying standard CORBA portable interceptors [24], as discussed in Section 5.3. Likewise, implementations based on the patterns [25] in the CORBA Component Model (CCM) [26] can implement load balancing without requiring changes to application code. In the CCM, a *container* is responsible for configuring the portable object adapter (POA) [4] that manages a component. Thus, TAO's adaptive on-demand load balancer just requires enhancing standard CCM containers so they support load balancing, without incurring other changes to application code.

3 The Design of the TAO CORBA Load Balancing Service

This section describes how we implemented a CORBA load balancing service in TAO [27, 5]. TAO is a CORBA-compliant [3] ORB that supports applications with stringent QoS requirements. TAO's 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. Below, we discuss the design challenges involved in developing TAO's load balancing service and present the solutions we applied to resolve them.

3.1 Overview

Before proceeding with the design discussion, we first describe the structure of the core components in TAO's load balancing service and then outline its dynamic interactions to illustrate how it operates.

3.1.1 The Component Structure of TAO's Load Balancing Service

Figure 5 illustrates the components in the TAO's load balancing service, which supports adaptive load balancing and on-demand request forwarding. Each of these components is outlined below:

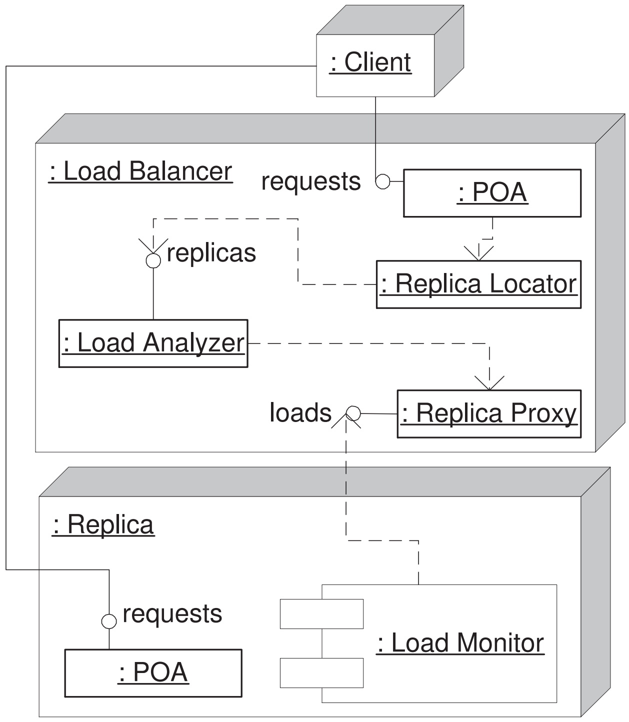

Figure 5: Components in the TAO Load Balancing Service

Replica locator: This component identifies which replicas will receive which requests. It is also the mechanism that binds clients to the identified replicas. The replica locator can be implemented portably using standard CORBA Portable Object Adapter (POA) mechanisms, such as servant locators [4], which implement the Interceptor pattern [25]. The Replica locator forwards each request it receives to the replica selected by the load analyzer described below.

Load monitor: This component (1) monitors loads on a given replica, (2) reports replica loads to a load balancer, and (3) responds to load advisories sent by the load balancer. As depicted in Figure 6, a load monitor can be configured with either of the following two policies:

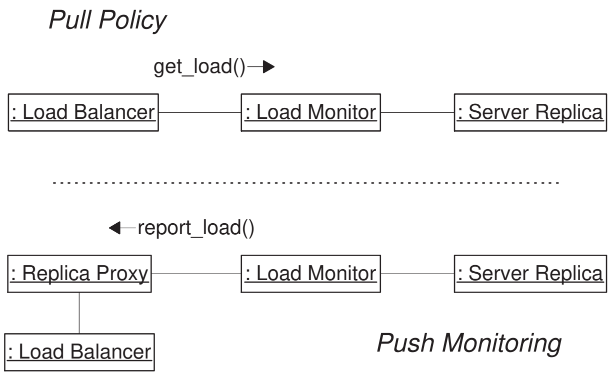

Figure 6: Load Reporting Policies

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

A load monitor also processes load advisories sent by the load balancer and informs replicas when they should accept requests versus forward them back to the load balancer.

Load analyzer: This component decides which replica will receive the next client request. The replica locator described above obtains a reference to a replica from the load analyzer and then forwards the request to that replica. 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, replica selection can be tailored to fit the dynamics of a system that is being load balanced.

Replica proxy: Each object managed by TAO's load balancing service communicates with it via a unique proxy [16]. The load balancer uses these replica proxies to distinguish different replicas because CORBA's so-called "weak" notion of object identity [20, 28], where two references to the same object may have different values.

Load balancer: This component is a mediator [16] that integrates all the components described above. It provides an interface through which load balancing can be administered, without exposing to clients the intricate interactions between the components it integrates.

3.1.2 The Dynamic Interactions in TAO Load Balancing Service

As described in Section 2.2, selecting the target replica using a non-adaptive balancing policy can yield non-uniform loads across replicas. In contrast, selecting a replica adaptively for each request can incur excessive overhead and latency. To avoid either extreme, therefore, TAO's load balancing service provides a hybrid solution via one of its load balancing strategies, whose interactions are shown in Figure 7. Each of the interactions in Figure 7 is described below.

1. A client obtains an object reference to what appears to be a replica and invokes an operation. In actuality, however, the client invokes the request on the load balancer transparently.
2. After the request is received from the client, the load balancer's POA dispatches the request to its servant locator, *i.e.*, the replica locator component.
3. Next, the replica locator queries its load analyzer for an appropriate server replica.

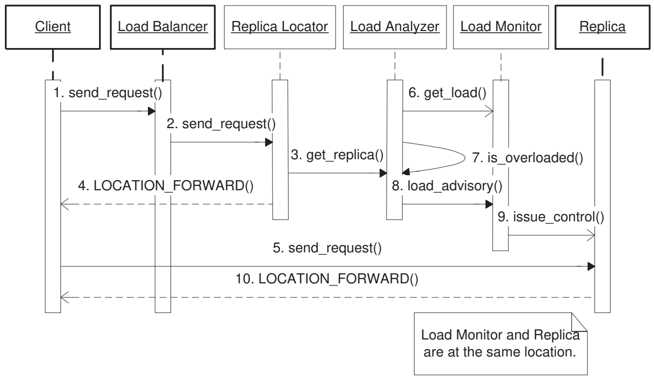

Figure 7: TAO Load Balancer Interactions

4. The replica locator then transparently redirects the client to the chosen replica.
5. Requests will continue to be sent *directly* to the chosen replica until the load balancer detects a high load on that replica. The additional indirection and overhead incurred by per-request load balancing architectures (see Section 2.2.2) is eliminated since the client communicates with the replica directly.
6. The load balancer monitors the replica's load. Depending on the load reporting policy (see *load monitor* description in Section 3.1.1) that is configured, the load monitor will either report the load to the balancer or the load balancer will query the load monitor for the replica's load.
7. As loads are collected by the load balancer, the load analyzer analyzes the load on the replica.
8. If a replica becomes overloaded the load balancer can dynamically forward the client invoking requests on it to another less loaded replica. To achieve the transparency requirements outlined in Section 2.1, TAO's load balancer does not communicate with the client application when forwarding it to another replica. Instead, TAO's load balancer issues a load advisory to the replica's load monitor.
9. The load monitor issues a load advisory message to the replica.
10. When instructed by the load monitor, the replica uses the GIOP LOCATION_FORWARD message to redirect the next request sent by a client back to the load balancer.
11. At this point the load balancing cycle starts again.

3.2 Design Challenges and Their Solutions

A number of design challenges were identified prior to and during the development of TAO's load balancing service. The challenges and the solutions applied to address them are discussed below. The solutions to each design challenge manifest themselves within the load balancing service components described in Section 3.1.1.

3.2.1 Challenge 1: Implementing Portable Load Balancing

Context: A CORBA load balancing service is being implemented in accordance with the requirements outlined in Section 2.

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 [25] 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 [3] features. For example, the role of the interceptor is played by a *servant locator*[4] and the role of the dispatcher is played by a *POA*. In particular, a *replica locator* can implement the standard CORBA `PortableServer::ServantLocator` [3] interface provided by the POA.

Figure 8 illustrates how load can be balanced transparently using standard CORBA features. Initially, clients are given an object reference to the load balancer, 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 replicas. Depending on the type of client binding granularity (see Section 2.2) selected by the application, one of the following actions will occur:

- The client will forward requests to the appropriate replica; or

[4]Servant locators allow CORBA server application developers to obtain custom object implementations dynamically, rather than using the POA's active object map [15].

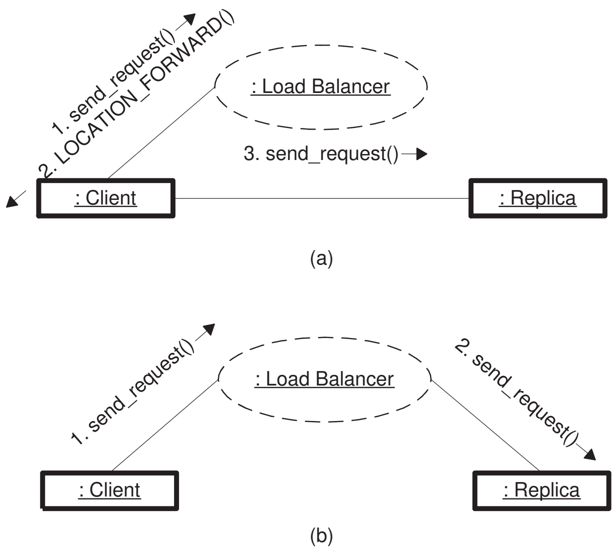

Figure 8: Load Balancing Transparency in Applications: (a) request forwarded by the client, (b) request forwarded on behalf of the client.

- The load balancer will forward requests to the appropriate replica on behalf of the client.

Applying the solution in TAO: In TAO, each replica registers itself with the load balancer. Each replica then becomes a potential candidate to handle a request intercepted by the load balancer. The interception is performed by a servant locator.

TAO's 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 [15]. Thus, the load balancer can use the servant locator to forward requests to the appropriate replica transparently, *i.e.*, without affecting server application code.

After receiving a request, the replica locator obtains a reference to the replica chosen by the load analyzer (see Section 3.2.4) and throws a `ForwardRequest` exception initialized with a copy of that reference. The server ORB constructs a LOCATION_FORWARD reply message from the `ForwardRequest` exception and returns it to the client. When the client ORB receives the LOCATION_FORWARD reply it forwards its original request transparently to the replica pointed at by the reference embedded within the reply.

3.2.2 Challenge 2: Load Balancing Heterogeneous Objects Efficiently

Context: The set of replicas being load balanced is called an "object group," which consists of replicas that implement the same IDL interface. Object groups are constructed when replicas of the same interface are registered with the load balancer. Depending on the replica membership policy in effect, an object group may be created by the load balancer or the distributed application itself, as follows:

- For application-controlled replica membership, applications are responsible for creating groups of object replicas.
- For infrastructure-controlled replica membership, the load balancer is responsible for creating object groups.

Infrastructure-controlled membership allows the addition of replicas to an object group transparently since no intervention is required by the server application.

Problem: If the load balancing service is to scale well, a single instance must be able to handle multiple different object groups. The same implementation of the load balancing service must be able to support both (1) multiple heterogeneous object groups, *i.e.*, that implement different interfaces and (2) multiple homogeneous object groups, *i.e.*, that implement the same interface. When binding a new client or rebinding an already bound client, the load balancing service must differentiate between the different object groups to ensure that each client request is forwarded to a replica with the appropriate interface.

For instance, a request intended for a replica that implements interface `A` should not be forwarded to a replica that implements interface `B`. Doing so would cause the request invocation to fail since the operation that is supposed to be invoked on a replica may not exist if the replica does not implement the expected interface. Worse yet, the operation may exist but its semantics may be entirely different or its arguments may not match. Depending on the POA implementation, an exception may be thrown or a catastrophic failure may occur. Even if the interfaces were identical, there can be significant semantic differences when invoking the same operation on different object groups, as each object group represents a different instance of a CORBA object.

So, how can we implement a load balancing service where each object group appears as a single object to the clients, while conserving memory and CPU resources in the load balancer implementation?

Solution → the Flyweight pattern: A *flyweight* [16] is a shared object that is indistinguishable from a non-shared instance of that object. It is useful for applications with a large number of objects. A flyweight's identity is logically equivalent to the identity of each of its constituents. As such, a single flyweight can be used in place of many distinct instances of objects with the same interface.

In the case of load balancing, an object group is represented as a flyweight. A client makes request invocations on that flyweight. The client has no idea it is issuing requests to a flyweight instead of a single distinct object. This design works because there is no logical difference in identity between the flyweight and its constituents, *i.e.*, the replicas in the object group represented by the flyweight. In essence, a flyweight provides a single identity by which the object group can be referenced.

Applying the solution in TAO: A CORBA object reference contains an `ObjectId` that identifies an object uniquely within a POA. Normally this information is used by the POA as a hash index or an active key into the active object map [15]. However, the application can configure the POA to define application specific semantics to the `ObjectId`, for example, an `ObjectId` can be used to identify an object group within a load balancer. The load balancer assigns a unique `ObjectId` to each object group. Each object group is implemented as a `sequence` of object references corresponding to the replicas within that object group.

TAO's load balancer constructs an object reference for each object group it load balances. When constructing the object reference for an object group, the group's `ObjectId` is embedded within the object reference using the `create_reference_with_id` operation defined in the `PortableServer::POA` interface. The generated object reference is a reference to the collective identity, *i.e.*, the flyweight, of the object group, as depicted in Figure 9. The cho-

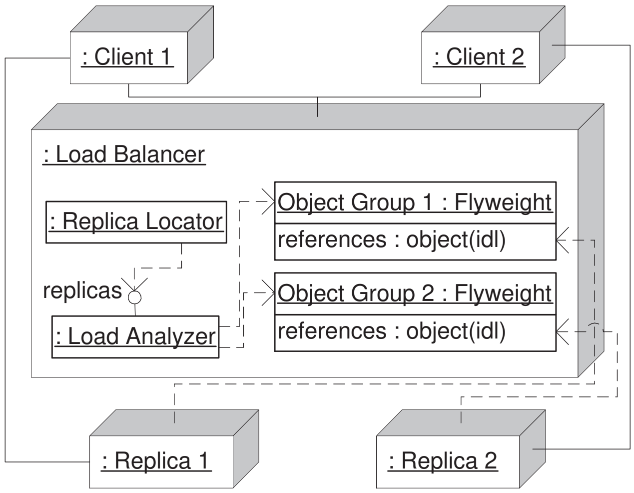

Figure 9: Representing a Group of Objects as a Flyweight

sen `ObjectId` plays a pivotal role in ensuring a given request is forwarded to a replica within an object group that implements the correct interface. This design also allows multiple object groups to implement the same interface.

Request forwarding occurs in the `preinvoke` method that is overridden from the `ServantLocator` base class. Among the arguments passed to this method by the POA is the `ObjectId` that was assigned to a given object group by the load balancer. The replica locator described in Section 3.2.1 uses the `ObjectId` to determine for which object group the request is intended. Since the `ObjectId` is unique within a given load balancer, it is straightforward to associate a request with an object group of the appropriate interface.

3.2.3 Challenge 3: Reacting to Feedback Flexibly and Administering Control Effectively

Context: An *adaptive* load balancing service must determine the current load conditions on replicas 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 replicas are balanced. These steps include (1) forcing the replica to redirect the client back to the load balancer when the load is high and (2) forcing the replica to once again accept client requests when the load is nominal.

Problem: Sampling loads from replicas should be as transparent as possible to the replicas. If load sampling was not transparent, a load balancer would have to sample loads from server replicas directly, which is undesirable since it would require replicas to collect loads. If replicas 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 was 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 replicas based on CPU, I/O, memory, network and battery power utilization.

In addition, a load balancer must react to various replica load conditions to ensure that loads across replicas are balanced. For example, when high load conditions occur, a replica must be instructed to forward the client request back to the load balancer so subsequent requests can be reassigned to a less loaded replica.

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 $\rightarrow$ the Strategy and Mediator patterns: The Strategy design pattern [16] 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. 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 a single load balancer can balance many different types of load.

The Mediator design pattern [16] defines an object that encapsulates how objects will interact. In addition to playing the role of a strategy, a load monitor also acts as a mediator between the load balancer and a given replica. This pattern ensures there is a loose coupling between the load balancer and the server replicas. Thus, the load balancer need not have any knowledge of the interface exported by the replica.

In its capacity as a mediator, a load monitor responds to load balancing requests sent by the load balancer. Depending on the type of request the load balancer sends the load monitor, the replica will either continue accepting client requests or redirect the client back to the load balancer. Note that the load balancer never interacts with the replica directly – all interaction occurs via the load monitor. Similarly, the replica never interacts with the load balancer directly. Instead, it interacts with the load balancer indirectly through the load monitor.

Applying the solution in TAO: When registering a replica with TAO's load balancer, its corresponding load monitor is also registered. As shown in Figure 10, the load balancer

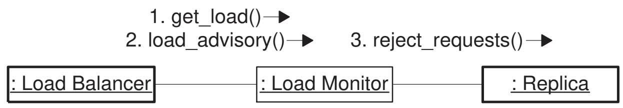

Figure 10: Feedback and Control when Balancing Loads

queries the load monitor for the load on the current replica, assuming that pull-based load monitoring is being used (see Section 3.1.1). In other words, the load balancer is getting *feedback* from the load monitor. Load balancing requests–called *load advisories*–are then sent to the load monitor from the load balancer and set the state of the current replica load to one of the following values:

- *Nominal* – When the load is nominal, the replica continues to accept requests.
- *High* – A high load advisory causes the replica to redirect client requests by forwarding them back to the load balancer, at which point the load balancer forwards the request to a less loaded replica.

This is where the load balancer administers *control*. TAO's load balancer is *adaptive* due to the bi-directional feedback/control channel between the load monitor and the load balancer.

Since the load monitor is decoupled from the load balancer it is possible to balance loads across replicas based on various types of load metrics. One type of load monitor could report CPU loads, whereas another could report I/O resource load. 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.

3.2.4 Challenge 4: Supporting Modular Load Balancing Strategies

Context: A distributed system employs a load balancing service to improve overall throughput by ensuring that loads across replicas 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. While in other applications the load cannot be easily predicted *a priori*.

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

In some applications it may even be necessary to change the load analysis algorithm dynamically, *e.g.*, to adapt to new application workloads. Moreover, bringing the system down to reconfigure the load balancing strategy may be unacceptable for applications with stringent 24×7 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. However, the load balancing service cannot simply implement all possible load balancing strategies, *e.g.*, application developers may wish to define application-specific or *ad-hoc* load balancing algorithms for testing.

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 → the Component Configurator pattern: The *Component Configurator* design pattern [25] allows applications to link and unlink components into and out of an application at run-time. In TAO's load balancing service this pattern can be used to change the replica selection strategy dynamically. Thus, a load balancer can use it to adapt to different load balancing use-cases, without being hard-coded to handle those use-cases.

At times it may be necessary to load balance only a few replicas, in which case a simple load balancing strategy may suffice. In other cases, 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 Component Configurator pattern makes it easy to dynamically configure load balancing algorithms appropriate for different use-cases without the need to stop and restart the load balancer.

Applying the solution in TAO: TAO's load analyzer uses the Component Configurator pattern to modularize the load balancing algorithm used when making load balancing decisions, as depicted in Figure 11. TAO's load balancing service

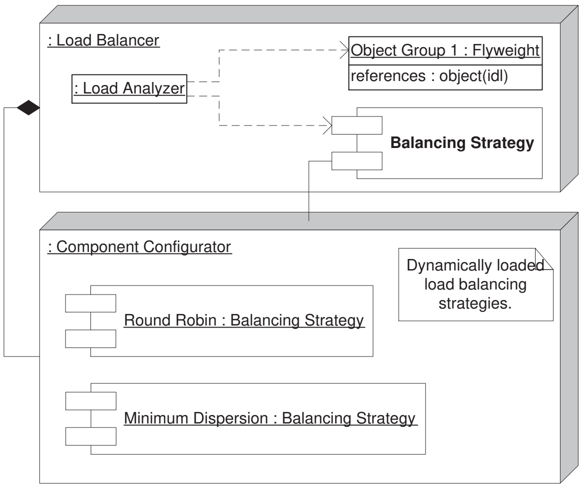

Figure 11: Applying the Component Configurator Pattern to TAO's Load Balancing Service

can be configured dynamically to support the following 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 replica in the object group being load balanced. Specifically, the next object reference in the sequence corresponding to that object group (see Section 3.2.2) is returned to the client. The client then uses the object reference returned by the load balancer to issue its request transparently to the actual replica.

After the object reference to the last replica is reached, the next request is forwarded to the first replica, at which point the cycle begins again. Using the terminology outlined in Section 2.2, this is a non-adaptive per-session load balancing strategy.

• **Minimum dispersion:** This adaptive strategy is more sophisticated than the round robin algorithm described in the preceding bullet. 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 replica is minimized. The steps used in this on-demand adaptive strategy are outlined below:

1. The average load across all replicas within a given object group is updated each time a load balancing decision occurs.
2. The instantaneous load on each replica is then compared to the average load.
3. If the difference between the average load and the instantaneous load is larger than the tolerance set by the *minimum dispersion* load balancing strategy, the load balancer will attempt to decrease the difference so that they fall within the tolerance.

Note that a set of replicas balanced via this strategy will not necessarily have the same load on each of them, but over time the load *dispersion* between the replicas will be minimized.

Recall from Section 3.2.3 that the load balancer and load analyzer are not load metric-specific. Although the load analyzer is not load metric-specific, it is only semantically meaningful to balance loads *within* object groups (see Section 3.2.2) rather than *across* object groups.

For example, imagine a situation where one object group is being balanced based on the average number of requests received per second and another object group is balanced based on the average CPU load. Since the type of load metric analyzed by the load balancer is opaque, the load balancer cannot know that the loads reported from the first object group differs from the same loads reported by the second object group. Thus, the average load magnitudes from both object groups could differ greatly. Attempting to balance loads across object groups in this situation would cause the load balancer to make incorrect balancing decisions. The only way to prevent this situation from occuring is to balance loads only within each object group, rather than across them.

3.2.5 Challenge 5: Coping With Adaptive Load Balancing Hazards

Context: Distributed applications employ adaptive load balancing strategies to improve overall system responsiveness.

Problem: Adaptive load balancing has the potential to improve system responsiveness. There are several hazards, however, that make it hard to ensure the stability of loads across replicas when the overall state of distributed systems changes quickly. These hazards are outlined below:

• **Thundering herd:** When a less loaded replica suddenly becomes available, a "thundering herd" phenomenon may occur if the load balancer immediately forwards all requests to that replica. If the rate at which the loads are reported and analyzed is slower than the rate at which requests are forwarded to the replica, it is possible that the load on that replica will increase rapidly. Ideally, the rate at which requests are forwarded to replicas 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 replicas, the harder it can be to balance loads across them successfully. For example, if only two replicas are available then one replica may be more loaded than the other. A naive load balancing strategy will attempt to shift the load to the less loaded replica, at which point it will most likely become the replica 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 overreacting transient, short lived or sample errors in the load metric?

Solution → Dampening load sampling rates and request redirection: The *minimum dispersion* load balancing strategy described in Section 3.2.4 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 replicas is balanced, a load balancer can adjust to large load discrepancies with much lower probability of experiencing the hazards discussed above. The criteria for deciding when to shift loads can also change dynamically as the number of replicas increases, in addition to the load.

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. TAO's minimum dispersion balancing strategy does not react to changes in load immediately because its default load balancing strategy averages instantaneous load samples with older load values. The empirical results presented in Section 4.3 illustrate the effects of TAO's dampening mechanisms.

3.2.6 Challenge 6: Identifying Objects Uniquely

Context: A load balancing service that manages multiple objects is responsible for collecting and analyzing information, such as the state, health, and environmental conditions, throughout the lifetime of each object it manages. This information is obtained from the load monitor, as described in Section 3.2.3. In some applications using a *pull model* to acquire the load information may not scale well and can be hard to optimize. In contrast, *push models* can resubmit load information when it has changed beyond a pre-set threshold or after a fixed period of time.

Problem: When receiving information about the load in one replica the load balancing service should determine the source of the load information efficiently and uniquely. This goal can be achieved easily via pull models, but it is harder to implement via push models. CORBA does not provide a lightweight mechanism to determine the source of a request.[5] Moreover, as described in Section 3.1.1, CORBA provides *weak identity* for objects, relying on the replica object reference to distinguish them would be not portable.

So, how can we portably and efficiently determine the source of the load information?

Solution → the Asynchronous Completion Token pattern: This pattern is used to efficiently dispatch processing tasks that result from responses to asynchronous operations invoked by a client [25]. In the load balancing service, the replica proxy plays the role of an asynchronous completion token (ACT). Load monitors communicate load updates via their replica proxy objects, as shown in Figure 12. The load balancing service creates a unique replica proxy for each monitor. When the replica proxy implementation is creates and caches the identity of the replica ACT and load monitor that will later use the replica proxy. This design allows the replica proxy to determine the identity of the remote replica efficiently whenever new load information is received.

Applying the solution in TAO: TAO uses a CORBA `Object` to play the role of an asynchronous completion token. The load balancing service creates a different CORBA

[5]The CORBA Security Service [29] can authenticate client requests, but this is a much more expensive mechanism that required for many applications.

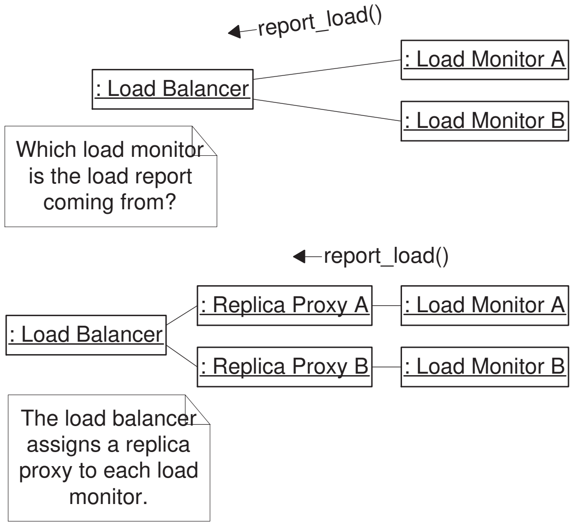

Figure 12: Identifying the Source of a Message Uniquely

`Object`–called a `ReplicaProxy`–for each replica. This proxy is created when the replica registers itself with the load balancing service initially. All future communication with the load balancing service is performed through the proxy. The Asynchronous Completion Token pattern allows the load balancing service to process the requests from each replica efficiently and unambiguously.

As each load is reported to the `ReplicaProxy`, the load analyzer is notified that a new load is available for analysis. Since the `ReplicaProxy` caches the object reference of its corresponding replica, the load balancer can redirect the client to a nominally loaded replica using the cached replica object reference.

3.2.7 Challenge 7: 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 replicas. Load monitors measure loads on replicas continuously and control client access to the replicas. Load analyzers decide if loads on replicas are nominal or high. Finally, replica locators bind clients to replicas.

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 implementation of distributed applications unduely, however, since certain functionality may be exposed to a given component unnecessarily. For instance, assume that a client communicates with the replica locator and load analyzer components directly. To obtain useful information, the client must be aware of how these components work, which violates the application transparency requirement discussed in Section 2.1. In such a case, shielding the client from such components is desirable.

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

Solution $\rightarrow$ the Mediator pattern: The Mediator pattern [16] 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.

The *load balancer* component ties together all the components listed above. It coordinates all interactions between other components, *i.e.*, it is a mediator. For example, it shields the client from the component interactions necessary to conduct load balancing. Thus, clients can remain unaware of the interactions mediated by the load balancer, which helps to satisfy application transparency requirements.

Applying the solution in TAO: As shown in Figure 5, the load balancer in TAO mediates the following types of component interactions:

• **Client binding interactions:** Rather than binding itself to a specific replica that may be highly loaded, TAO's load balancer binds the client to a suitable replica. The load balancer creates an object reference that corresponds to a group of replicas–called an *object group*–being load balanced. Instead of using an object reference that directly refers to a given replica, 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 replica chosen by the load balancer.

It is important to note that the CORBA object model was intentionally designed to decouple the object implementation from the object references that clients use to access the implementations. In TAO's load balancing service we exploit this feature of CORBA to hide the particular location, number, and characteristics of the replicas behind an object reference that points clients to the load balancing service. Clients applications are shielded by this extra level of indirection by their ORBs, and use a load balanced object just like any other CORBA object, unaware of the situation except perhaps for the difference in performance. The Flyweight pattern allows us to implement multiple objects efficiently, as described in Section 3.2.2.

The load balancer also rebinds the client to another replica by using other components, such as the load monitor. 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. 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 those loads were obtained is hidden from the load analyzer.

4 Performance 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 load balancing service described in this paper is that it increases overall system throughput by distributing requests across multiple back-end servers (replicas) *without* increasing round-trip latency and jitter significantly.

This section describes the design and results of several experiments that were 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 2.2. The first set of experiments in Section 4.2 show the amount of overhead incurred by the request forwarding architectures described in this paper. The second set of experiments in Section 4.3 demonstrate how TAO's load balancer can maintain balanced loads dynamically *and* efficiently, whereas alternative load balancing strategies cannot.

4.1 Hardware/Software Benchmark 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 13. All benchmarks were run in the POSIX real-time thread scheduling class [30]. 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.[6] Figure 1 illustrates the basic design

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

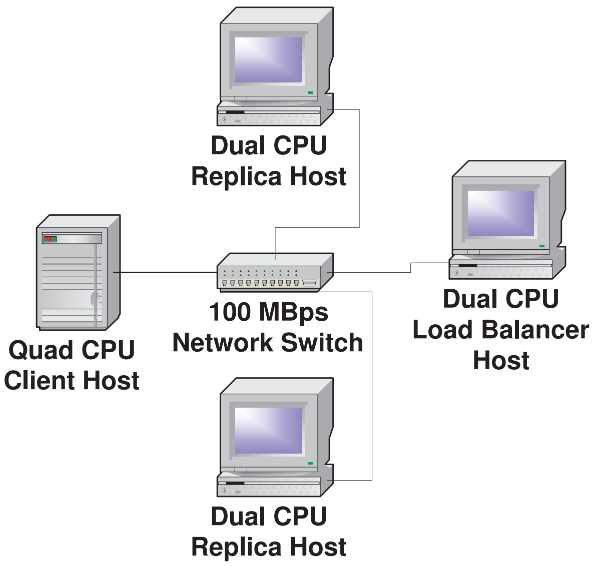

Figure 13: Load Balancing Experiment Testbed

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 request mades 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 can run 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 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 adds support for TAO's adaptive on-demand load balancer to the classic `Latency` test. The `Latency` test client code remained unchanged, so client transparency was preserved. This variant quantified the performance and scalability impact of using TAO's adaptive on-demand load balancer.

4.2 Benchmarking the Overhead of Load Balancing Mechanisms

Overhead benchmarks measure the degree of end-to-end overhead incurred by adding load balancing to a CORBA application. 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 using the following four request forwarding architectures:

1. Without load balancing, *i.e.*, where the client always communicates with the same server directly.
2. With the "non-adaptive per-session" client binding granularity, where the load balancer only forwards the client to a replica once. The client continues sending requests to that replica for the remainder of its lifetime.
3. With the "non-adaptive per-request" client binding granularity, where the load balancer forwards requests to replicas on behalf of clients. No load feedback is sent to the load balancer, however.
4. With TAO's "adaptive on-demand" client binding architecture, where the client is forwarded to another less loaded replica chosen by the load balancer when it detects load imbalance.

Overhead measurement technique: All overhead benchmarks were run with 200,000 iterations. To establish a performance baseline, the `Latency` performance test was first run between a single-threaded client and a single-threaded server residing on separate workstations. These results reflect the baseline performance of a TAO client/server application.

Afterwards, we 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.

Next, we added a specialized non-adaptive per-request "forwarding server" to the original `Latency` test. The sole purpose of this server is to forward client requests to an unmodified server. The forwarding server resided on a different machine than either the client or server, which themselves each ran on separate workstations. Since the forwarding 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.

Finally, TAO's adaptive on-demand client binding granularity was included in the experiment by adding the *load monitor* described in Section 3.2.3 to the `Latency` test server. This enhancement made it possible for 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 14 quantify the latency imposed by adding load balancing, specifically request forwarding to the `Latency` performance test. As shown in this figure, a non-adaptive per-session approach imposes essentially no latency overhead to the classic `Latency` test. The non-adaptive per-request approach, on the other hand, 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 (1) the additional processing resources the load monitor needs to perform load monitoring and (2) 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 maintain low latency performance, and still provide adaptive load balancing.

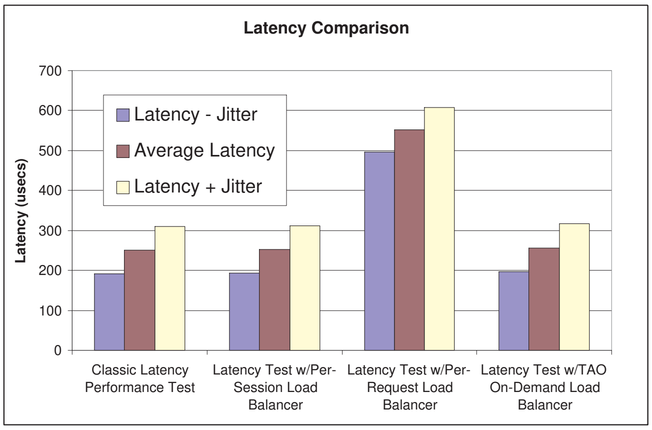

Figure 14: Load Balancing Latency

Figure 15 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 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 15 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.

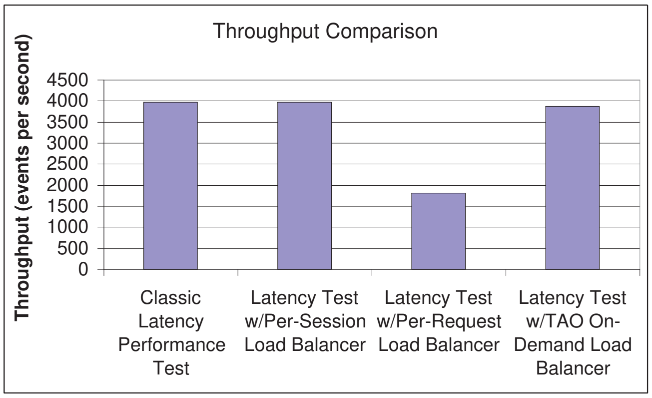

Figure 15: Load Balancing Throughput

In contrast, the throughput in TAO's load balancing approach decreased only 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 use of resources necessary to facilitate communication between the load balancer and the load monitor.

4.3 Load Balancing Strategy Effectiveness

The following set of benchmarks quantify the how effective each of the load balancing strategies 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 4.2 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.

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 were configured to issue 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. 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 16 illustrates the loads incurred on each of the `Latency` server replicas using non-adaptive per-session load

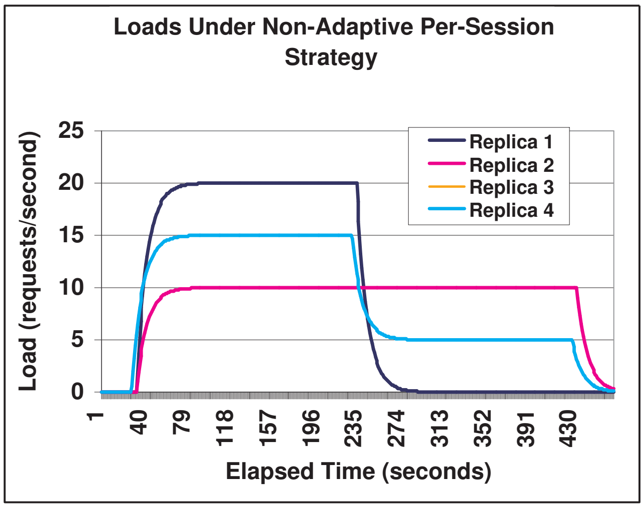

Figure 16: Effectiveness of Non-Adaptive Per-Session Load Balancing

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 is 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 policy effectiveness:** This test focused on demonstrating the benefits of an adaptive load balancing strategy. Therefore, we increased the load imposed by each client and increased the 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. The same testbed used for the above experiments was used for this experiment.

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 larger client requests rates. If the number of iterations were kept at the 200,000 used in the above overhead experiments, then it is possible that this experiment may have ended before loads across the replicas were balanced.

As Figure 17 illustrates, the loads across all four replicas

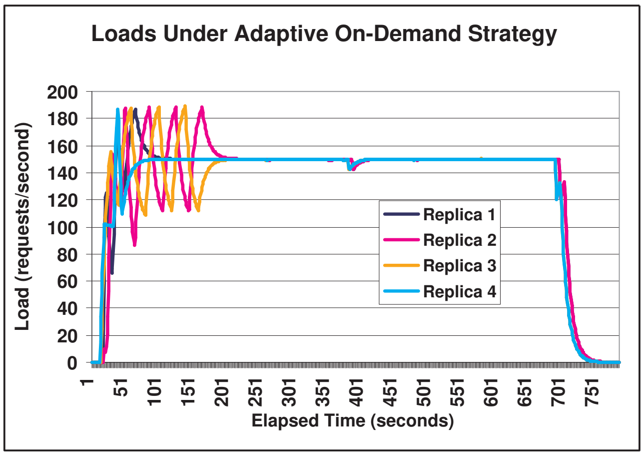

Figure 17: Effectiveness of Adaptive On-Demand Load Balancing

fluctuated for a short period of time until an equilibrium load of 150 Hz was reached.[7] The initial load fluctuations result from the load balancer periodically rebinding clients to less

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

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 3.2.5.

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 3.2.5) built into TAO's adaptive on-demand load balancing strategy, it is likely that the 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 forces it use average loads instead. Compare these results to the non-adaptive per-session load balancing architecture results in Figure 16. 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 17. Contrast this to the non-uniform drops in load that occured in the overhead experiments in Section 4.2, 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.

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 3.2.4. 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, the point is that an adaptive load balancing strategy, such as the one demonstrated by this experiment, can maintain balanced loads across a given set of replicas.

5 Related and Future Work

This section outlines related research on load balancing and describes how it compares and contrasts to our work on TAO's load balancing service. We first compare our middleware-based load balancing strategies with work at other levels of abstractions. Next, we describe how our work compares with other research on CORBA-based load balancing. We finish by outlining our future research plans to further enhance TAO's load balancing service.

5.1 Related Research on Load Balancing

As discussed in Section 1, load balancing mechanisms have been implemented at various levels, such as in the network, OS, and middleware. Some implementations, such as the Condor [31] and Beowulf [32] clustering systems combine aspects from multiple levels. This paper focuses primarily on middleware-based load balancing, but many concepts and patterns used in middleware-based load balancing also apply to network-based and OS-based load balancing, as described below.

Network-based load balancing: Network-based load balancing implementations often make decisions based on the frequency with which a given location is accessed. The decision of where to service a request can be made at various stages along the path to its destination. For example, a router or DNS server could decide where to send a request.

Network-based load balancing has the disadvantage that load balancing decisions are based solely on the request target, which hampers flexibility greatly. However, recent developments in network-based load balancing do take advantage of request content. These hybrid implementations [1] provide finer-grained load analysis, which can improve load balancing decisions. Nevertheless, the choice of metric used in load balancing decisions is still restricted to the frequency with which a given target is accessed.

Unfortunately, frequency alone is not always an adequate load metric since some requests may incur large loads on the target host. For instance, such scenarios can occur when servicing CGI requests. When combined with load balancing decisions based solely on target access frequency, the increased loads from such requests can degrade overall system performance. It is possible to analyze the content of each request to determine if it is a "high load" request, but this requires *a priori* knowledge of the requests behavior, which may not be feasible in many distributed computing systems.

OS-based load balancing: Some distributed operating systems, such as Chorus [33], can distribute processes transparently across remote OS endsystem nodes. Load balancing performed at this level has the advantage that it can be implemented transparently to applications. When loads are too high at their current location, running applications can be migrated to other nodes relatively transparently. As with the network-based architecture, however, load balancing at this level makes it hard to choose which metric to use when deciding where to move processes since application-level metrics and policies are not available at this level.

Middleware-based load balancing: Middleware-based load balancing implementations sit between the application and the OS/network. Middleware shields the application from tedious and error-prone low-level OS complexities, while also providing a powerful interface to make load balancing at this level as transparent as possible, if not completely transparent. Moreover, middleware can be implemented with sufficient flexibility to overcome the disadvantages in network-based and OS-based load balancing architectures.

CORBA is a prime example of a technology that provides the following capabilities needed to implement an effective load balancing service:

- Application developers can customize how their system is load balanced without being restricted by the limited– and often hard-coded–metrics available in network-based and OS-based load balancing.
- Applications can select at run-time the metric(s) used in load balancing decisions.
- New metrics can also be defined with relative ease by separating interface from implementation, *i.e.*, exposing a consistent interface for each implementation. In particular, Section 3.2.3 describes how a load monitoring component can be implemented for a specific load metric, yet keep the load balancing service load metric neutral.

Moreover, a middleware-based load balancing service can be used in conjunction with network- and OS-based load balancing facilities, which supports some interesting load balancing combinations. For example, if all an application needs is balancing based on request frequency then a middleware-based load balancer can delegate load balancing tasks to the network or OS layers. Another possibility is that the middleware-based load balancer itself could be load balanced. Since those load balancing tasks would be performed at the network or OS level, additional resources are available for use by the middleware-based load balancer and other applications.

Other examples of middleware-based load balancing include some Web server implementations. Web servers can forward HTTP requests to any one of a number hosts that replicate the target web page. Overall throughput can be increased using any of the load balancing approaches detailed in this paper. The key is that the web server performs the HTTP request load balancing.

The CORBA-based load balancing concepts detailed in this paper are generally applicable to other middleware implementations, such as COM+. In fact, a middleware-based load balancing service called *Component Load Balancing* (CLB) is available from Microsoft for COM+ applications.

5.2 Related Research on CORBA-based Load Balancing

CORBA load balancing can be implemented at several levels in the OMG reference architecture, such as the following:

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 [34]. For example, Inprise's VisiBroker implements a similar strategy, where Visibroker's Portable 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 the load balancing mechanism is closely coupled with the ORB. For example, the length of communication paths is shorter. 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 greatly impedes 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 or Portable 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 [34], TAO's approach is transparent to clients. Unlike the ORB-based approaches, however, our implementation is based only in 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 [35] 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 keep the consumers distributed evenly. In contrast, TAO's load balancing service can be used for application defined objects, not only event services.

Various commercial CORBA implementations also provide service-level load balancing. For example, IONA's Orbix [36] can perform load balancing using the CORBA Naming Service. Different replicas are returned to different clients when they resolve the object. This is a typical non-adaptive per-session load balancer, which suffers from the disadvantages described in Section 2.2. BEA's WebLogic [37] uses a per-request load balancing strategy, as described in Section 2.2. 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 the Orbix load balancing service.

5.3 Future Work

This paper addresses an important subset of the load balancing service design space. In particular, we focus on client transparency, centralized load balancing, stateless replicas, and "best-effort" adaptive balancing policies. However, additional steps can be taken to enhance TAO's load balancing service. For example, a complete solution should also provide other capabilities, such as supporting (1) transparent load balancing to server object replicas, (2) decentralized load balancing models, (3) distributed systems whose servers retain state, (4) enhanced server side scalability, (5) fault tolerant load balancing, and (6) support for better quality of service. Below, we outline future work that we are conducting to address these topics.

Server transparency: It is non-trivial to achieve transparent server load balancing since obtaining feedback from a given replica and controlling it without altering server application code is hard. Fortunately, CORBA-based distributed systems can achieve server transparency by taking advantage of the following recently standardized CORBA features:

- **Portable Interceptors:** Portable interceptors [24, 38] the transparent capture of client requests before they are dispatched to the replica. For example, a *server request interceptor* could be added to the ORB where a given replica runs. Since interceptors reside within the ORB no modifications to server application code are necessary, other than registering the interceptor with the ORB.

- **CORBA Component Model (CCM):** The CCM [39] introduces *containers* to decouple application component logic from the configuration, initialization, and administration of servers. In the CCM, a container creates the POA and interceptors required to activate and control a component. These are the same CORBA mechanisms used to implement the server components in TAO's load balancing service. The standard CCM containers can be extended to implement automatic load balancing *generically* without changing the application logic of components.

Decentralized load balancing models: The CORBA-based load balancing architecture described in this paper is based on a *centralized* load balancing model. Specifically, it assumes that one load balancer performs all load balancing tasks for a particular distributed system. Such a model has the advantage that it simplifies the design and implementation of the load balancer, but it also introduces a single point of failure, which can impede system reliability.

One solution is to implement a *cooperative* load balancing service. In this model, load balancing is facilitated through a distributed set of load balancers that collectively form a single *logical* load balancing service. This model has the advantage that a single point of failure does not exist, and that no single bottleneck point exists either. Load balancing decisions would be made cooperatively, *i.e.*, each load balancer could communicate with other balancers to decide how best to balance loads across a given group of replicas.

Stateful replicas: Another issue that we are covering in future work involves load balancing of stateful replicas. To load balance replicas that retain state, some means of maintaining state consistency between replicas is necessary. Typical techniques to achieve this consistency include (1) using reliable multicast to share the current state efficiently between multiple replicas, (2) providing hooks within a replica that allow a load balancer to perform state transfers explicitly to another less loaded replica so that request servicing can continue there, or (3) a combination of both (1) and (2). Efficient load balancing of stateful replicas is non-trivial, however, due to the additional load incurred by ensuring state consistency between replicas.

Load monitoring granularity: A server can have multiple objects running in it. If there are a large number of objects in the server then instantiating a load monitor (see Section 3.2.3) for each object may not scale. For example, load monitor resources, such as memory, CPU, network bandwidth, can starve objects or processes running on the same server.

To improve the scalability of the load balancing system, we plan to support more scalable load monitoring granularities. Rather than instantiating a load monitor for each object on the server, a single load monitor could be associated with a group of objects that share a common load metric. For example, despite the fact that objects may implement different interfaces, all are load balanced based on CPU utilization.[8]

We believe this design can significantly reduce the amount of resources imposed by adding the server load balancing support, *i.e.*, load monitors for a large number of objects residing in the same server. However, it also introduces some complexities to the load monitor implementation. For example, suppose that the load balancer detects a high load and issues a load advisory to the shared load monitor. The load monitor now must decide which objects sharing that load monitor should shed their load, *e.g.*, by forcing the client to contact the load balancer so that it can be re-bound to another replica.

Fault tolerant load balancing: By using the adaptive CORBA-based load balancing architecture described in this

[8] This should not be confused with the load balancing of multiple object groups discussed in Section 3.2.2. Here, several objects residing in the same server are sharing a single load monitor, as opposed to the load balancer ensuring requests are sent to a replica that implements the correct interface.

paper, it is still possible for clients that have not been forwarded to replicas to be denied service. Therefore, some form of fault tolerance is needed to prevent this situation. Fortunately, CORBA defines a standard *Fault Tolerance* [20] service to address these types of failures.

Making a load balancing service fault tolerant by means of a Fault Tolerance CORBA service can alleviate one of the inherent problems with centralized load balancing: its single point of failure. It can also ensures that state within replicas is consistent, in the case of stateful replicas. This capability can simplify a load balancer implementation since the load balancer can delegate the task of ensuring state consistency between replicas to the Fault Tolerance service. One implementation of the CORBA Fault Tolerance service is DOORS [12, 40]. Since it is itself a CORBA service and is implemented using TAO, integration of DOORS into TAO's load balancer should be straightforward.

Improved quality of service support: As mentioned in Section 3.2.5, it is hard to ensure that loads across replicas stay balanced evenly when the overall state of distributed systems changes rapidly. For example, several new replicas may be added to an object group dynamically, which cannot be predicted by a load balancer. Likewise, a poorly designed load balancing strategy cannot handle degenerate load balancing conditions, such as unstable replica loads.

Several approaches can be used to improve the effectiveness of a given load balancing strategy:

- Take into account past load trends in an effort to anticipate future load conditions.
- Take advantage of sophisticated algorithms based on control theory that are designed specifically to restore system equilibrium when being 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.

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

6 Concluding Remarks

As the Internet becomes more pervasive and applications become more distributed, the demand for greater scalability and dependability 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 the load across object replicas 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 replicas, 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 [5]. The following is a synopsis of our lessons learned as a result of this work:

1. Though CORBA provides solutions for several distributed system challenges, such as security, transactions, and fault tolerance, it still lacks standard solutions to tackle many other difficulties commonly faced by distributed systems architects and developers. Chief among those missing facilities are load balancing, state caching, and state replication. When such higher-level services are available, developers can concentrate on their core application behavior, rather than wrestling with complex infrastructure mechanisms needed to make their application distributed, scalable, and dependable.

2. The CORBA-based load balancing service provided by TAO fills part of the gap that allows distributed applications to be load balanced adaptively and efficiently. TAO's load balancing service increases overall system throughput by distributing requests across multiple back-end server replicas *without* increasing round-trip latency substantially or assuming predictable, or homogeneous loads.

3. 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 applications, including many telecommunication systems, aerospace systems, online trading systems, medical systems, and manufacturing process control systems. All the source code, examples, and documentation for TAO is freely available from URL `www.cs.wustl.edu/~schmidt/TAO.html`.

References

[1] E. Johnson and ArrowPoint Communications, "A Comparative Analysis of Web Switching Architectures." http://www.arrowpoint.com/solutions/white_papers/ws_archv6.html, 1998.

[2] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*. Harlow, England: Pearson Education Limited, 2001.

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

[4] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1999.

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

[6] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, to appear 2000.

[7] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[8] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[9] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[10] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, To appear 2000.

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

[12] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. 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), OMG, Sept. 2000.

[13] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[14] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[15] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

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

[17] D. C. Schmidt, V. Kachroo, Y. Krishnamurthy, and F. Kuhns, "Applying QoS-enabled Distributed Object Computing Middleware to Next-generation Distributed Applications," *IEEE Communications Magazine*, vol. 20, Oct. 2000.

[18] 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.

[19] L. Moser, P. Melliar-Smith, and P. Narasimhan, "A Fault Tolerance Framework for CORBA," in *International Symposium on Fault Tolerant Computing*, (Madison, WI), pp. 150–157, June 1999.

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

[21] M. Henning, "Binding, Migration, and Scalability in CORBA," *Communications of the ACM special issue on CORBA*, vol. 41, Oct. 1998.

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

[23] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[24] Adiron, LLC, *et al.*, *Portable Interceptor Working Draft – Joint Revised Submission*. Object Management Group, OMG Document orbos/99-10-01 ed., October 1999.

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

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

[27] Center for Distributed Object Computing, "TAO: A High-performance, Real-time Object Request Broker (ORB)." www.cs.wustl.edu/~schmidt/TAO.html, Washington University.

[28] M. L. Powell, *Objects, References, Identifiers, and Equality White Paper*. SunSoft, Inc., OMG Document 93-07-05 ed., July 1993.

[29] Object Management Group, *Security Service 1.8 Specification*, OMG Document security/00-09-01 ed., September 2000.

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

[31] J. Basney and M. Livny, "Deploying a High Throughput Computing Cluster," *High Performance Cluster Computing*, vol. 1, May 1999.

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

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

[34] M. 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), OMG, Sept. 2000.

[35] K. S. Ho and H. V. 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), OMG, Sept. 2000.

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

[37] BEA Systems Inc., "WebLogic Administration Guide." http://edoc.bea.com/wle/.

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

[39] N. Wang, D. C. Schmidt, and C. O'Ryan, "An Overview of the CORBA Component Model," in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, MA: Addison-Wesley, 2000.

[40] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. 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), ACM/IEEE, Dec. 2000.

[41] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

A Overview of the CORBA Reference Architecture

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [4]. Figure 18 illustrates the key components in the CORBA reference model [3] that collaborate to provide this degree of portability, interoperability, and transparency.[9] Each component in the

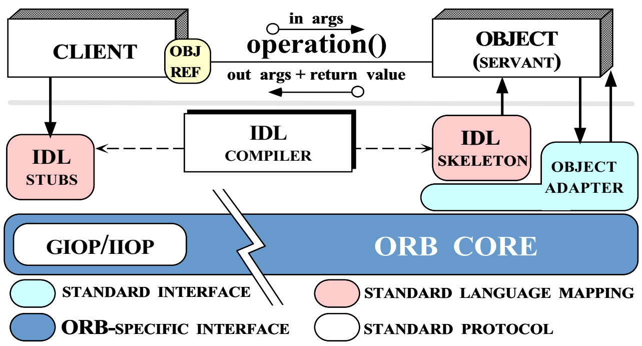

Figure 18: Key Components in the CORBA 2.x Reference Model

CORBA reference model is outlined below:

Client: A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object→operation(args)`. Figure 18 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object reference*, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern [16] and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern [16] and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [41].

[9] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA's components see [3].

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.