The Design 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 *

February 13, 2001

A subset of this paper will appear in the Distributed Systems Engineering Journal's "Online" edition, April 2001.

Abstract

As network-centric computing 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 middleware mechanisms can be used to distribute system load equitably across object replicas residing on multiple servers. This paper describes 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.

Keywords: Middleware, patterns, CORBA, load balancing.

1 Introduction

An increasingly popular and cost effective technique to improve networked server performance is *load balancing*, where hardware and/or software mechanisms distribute client workload equitably among back-end servers to improve overall system responsiveness. This paper focuses on middleware-based load balancing supported by CORBA [1] *object request brokers* (ORBs). ORB middleware allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware [2]. Moreover, ORBs can determine which client requests to route to which object replicas on which servers.

Our previous research on middleware has examined many dimensions of ORB endsystem design, including static [3] and dynamic [4] scheduling, event processing [5], I/O subsystem [6] and pluggable protocol [7] integration, synchronous [8] and asynchronous [9] ORB Core architectures, ORB fault tolerance [10], systematic benchmarking of multiple ORBs [11], patterns for ORB extensibility [12], ORB performance [13], and CORBA load balancing performance [14]. This paper focuses on another dimension in the CORBA research domain: *the design 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 [1] (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's rich set of features provides the means to realize an adaptive load balancing service. CORBA is an effective choice for distributed systems due to the inherent distribution and common heterogeneity of clients and servers written in different programming languages running on different hardware and software platforms. In this context, CORBA can simplify system implementation because it offers a language- and platform-neutral communication infrastructure. Moreover, it reduces development effort by offering higher level programming abstractions that shield application developers from distribution complexities, thereby allowing them to concentrate their efforts on stock trading business logic.

The remainder of this paper is organized as follows: Section 2 summarizes the requirements of CORBA-based load balancing services; Section 3 describes the design of our load balancing service, which is based on standard CORBA features and implemented using the TAO open-source[1] CORBA-compliant ORB; Section 4 outlines the key challenges we faced when design TAO's load balancing service and illustrates the patterns we applied to address these challenges; and Section 5 presents concluding remarks.

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

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

2 Requirements for a CORBA Load Balancing Service

The OMG CORBA specification provides the core capabilities needed to support load balancing. In particular, a CORBA load balancing service can take full advantage of the *request forwarding* mechanism[2] mandated by the CORBA specification [1]. A CORBA server application can use this mechanism to 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 load balancing model: In the CORBA programming model objects are the unit of abstraction and system architects reason about objects in order to manage their available resources. Thus, the granularity of load balancing in CORBA should be based on objects, rather than, *e.g.*, processes or TCP/IP addresses. Moreover, a load balancing service and ORB should coordinate the interactions amongst *multiple* object replicas. Sets of multiple object replicas are called *object groups* or *replica groups*.

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

Server application transparency: Although load balancing should ideally require few modifications to servers, this goal is hard to achieve in practice. For example, load balancing a stateful CORBA object requires the transfer of its state to a new replica. The application implementation must either perform the transfer itself or define hooks that allow the load balancing framework to perform the state transfer as unobtrusively as possible [15].

The situation for stateless CORBA servers is different. In this case, the implementation of an server 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, collecting these metrics may require some modifications to server application code.

Dynamic client operation request patterns: Load balancing services can be based on various client request patterns. For example, load balancers for certain types of systems assume client requests occur at deterministic or stochastic rates that execute for known or fixed durations of time. While these assumptions may apply for certain types of applications, such as continuous multimedia streaming [16], they do not apply in complex Internet or military [17] environments where client operation request patterns are dynamic and the duration of each request may not be known in advance. In this paper, therefore, we focus on load balancing techniques that do not require *a priori* scheduling information.

Maximize scalability and equalize dynamic load distribution: Although it is common practice to design lightweight load distribution capabilities, *e.g.*, based on extensions to naming services [18], these approaches do not balance dynamic loads equitably, which limits their scalability. Thus, a CORBA load balancing service must increase system scalability by maximizing dynamic resource utilization in a group of servers whose resources would not otherwise be used as efficiently. By improving resource utilization via load balancing, the overall scalability of the server group should be enhanced significantly.

Increase system dependability: Load balancing services can also handle certain types of server failures. By using administrative interfaces or automated policies, for example, 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 should not be 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 the removal of replicas for upgrades, preemptive maintenance, or re-allocation of system resources.

Minimal overhead: A CORBA load balancing service should not introduce undue latency or networking overhead since otherwise it can actually reduce–rather than enhance–overall system performance. In particular, an implementation

[2] The standard CORBA LOCATION_FORWARD GIOP message used to facilitate this request forwarding mechanism is discussed in Section 4.0.1.

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

that (1) increases the average number of messages per-request or (2) uses a single server to process all requests may be inappropriate for high-performance and/or large-scale applications. [30] 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.

3 The Design of TAO's Load Balancing Service for CORBA

This section describes the design of an adaptive load balancing service in TAO [3], which is a CORBA-compliant ORB that supports applications with stringent QoS requirements. TAO's load balancing service is designed to support the requirements presented in Section 2.

3.1 Component Structure in TAO's Load Balancing Service

Figure 1 illustrates the components[4] in the TAO's load balancing service, which supports adaptive load balancing and on-demand request forwarding. Each of these components is outlined below:

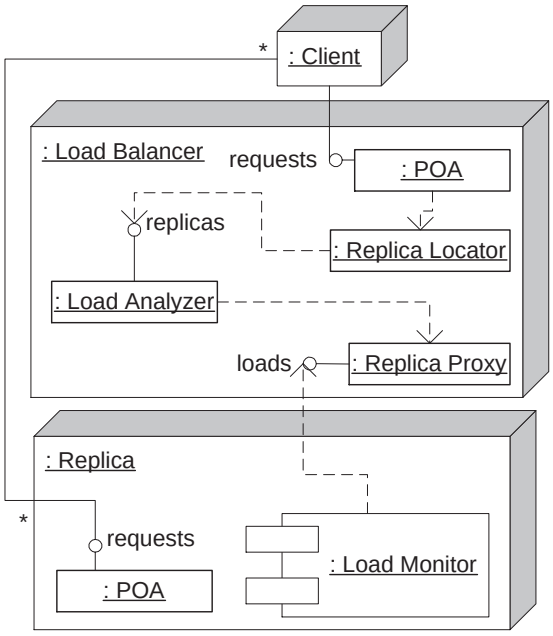

Figure 1: Components in the TAO Load Balancing Service

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 [2], which implement the Interceptor pattern [22]. 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 2, a load monitor can be configured with either of two policies:

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

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

[4]The term *component* used throughout this paper refers to a "component" in the general sense, *i.e.*, an identifiable entity in a program, rather than in the more specific sense of the CORBA Component Model [21].

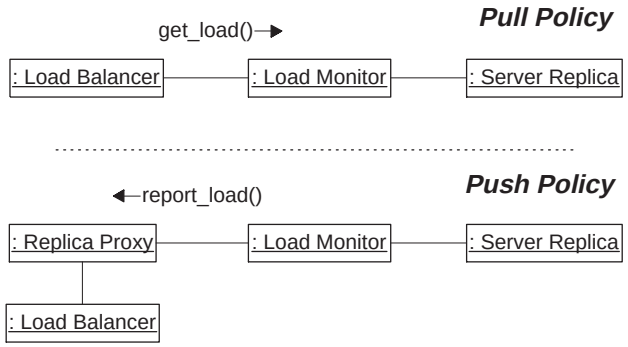

Figure 2: Load Reporting Policies

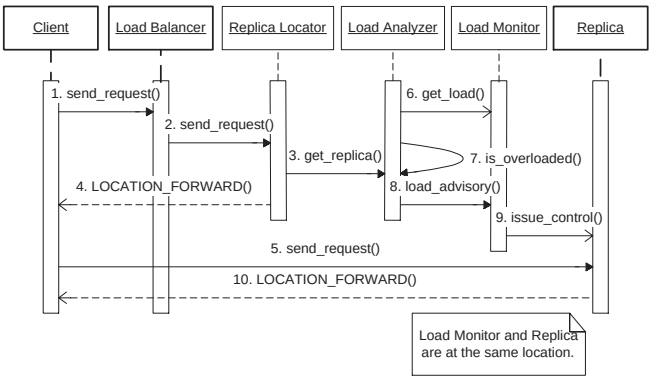

Figure 3: TAO Load Balancer Interactions

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. The load balancer uses these replica proxies to distinguish different replicas to workaround CORBA's so-called "weak" notion of object identity [20], where two references to the same object may have different values. Thus, it is only possible to compare the *equivalence* of two object references. Two object references are equivalent if they refer to the same object. Otherwise, they are not equivalent if they do not refer to the same object or the ORB was unable to make this determination. It is the intentional ambiguity of the latter case that makes CORBA object identity "weak."[5] Section 4.0.5 discusses the replica proxy in more detail.

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

3.2 Dynamic Interactions in TAO's Load Balancing Service

As described in [30], selecting a 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 3. Each interaction in Figure 3 is outlined below.

1. A client obtains an object reference to what appears to be a replica and invokes an operation. In actuality, however, the client transparently invokes the request on the load balancer itself.
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.
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 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) 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 to another less loaded replica. To achieve the transparency requirements outlined in Section 2, 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.

[5] See [23] for the rationale behind CORBA's object identity semantics.

9. The load monitor issues a control message to the replica. Depending on the contents of the load advisory issued by the load balancer, this control message will cause the replica to either accept or redirect requests.
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.

4 Design Challenges and Their Solutions

The following design challenges were identified prior to and during the development of TAO's load balancing service:

1. Implementing portable load balancing
2. Enhancing feedback and control
3. Supporting modular load balancing strategies
4. Coping with adaptive load balancing hazards
5. Identifying objects uniquely
6. Integrating all the load balancing components effectively

These challenges and the solutions we 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.

4.0.1 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 [22] 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 [1] features. For example, the role of the interceptor is played by a *servant locator*[6] and the role of the dispatcher is played by a *POA*. In particular, a *replica locator* can implement the standard CORBA `ServantLocator` [1] interface provided by the POA.

Figure 4 illustrates how load can be balanced transparently using standard CORBA features. Initially, clients are given

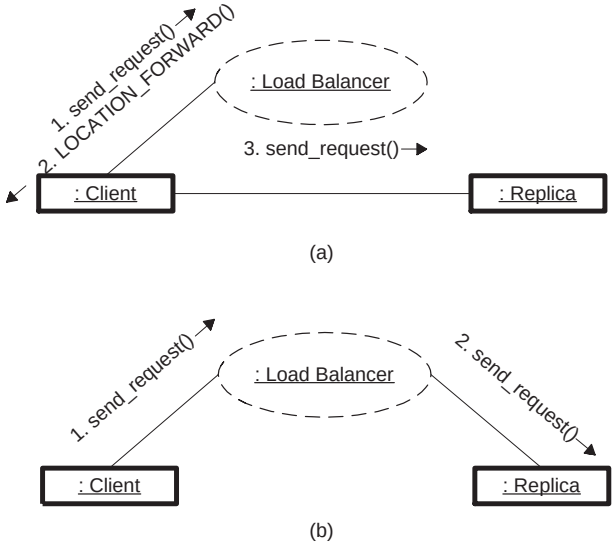

Figure 4: Load Balancing Transparency in Applications: (a) request forwarded by the client and (b) request forwarded on behalf of the client.

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 selected by the application, one of the following actions will occur:

- The client will forward requests to the appropriate replica, as shown in Figure 4(a); or
- The load balancer will forward requests to the appropriate replica on behalf of the client, as shown in Figure 4(b).

[6] Servant locators are a meta-programming mechanism [24] that allows CORBA server application developers to obtain custom object implementations dynamically, rather than using the POA's active object map [13].

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 [13]. 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 4.0.3) and throws a `ForwardRequest` exception initialized with a copy of that reference. The server ORB catches this exception and then returns a LOCATION_FORWARD GIOP message. When the client ORB receives this message, the CORBA specification requires it to

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

Thus, a server application and an ORB can forward client requests to other servers *transparently*, *portably*, and *interoperably*.

4.0.2 Challenge 2: Enhancing Feedback and Control

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 its load is high and (2) forcing the replica to once again accept client requests when its 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 were load-metric specific it would be costly to deploy load balancers for distributed applications that require balancing based on several load metrics. For example, a separate load balancer would be needed to balance replicas based on various metrics, such as 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 [25] design pattern allows the behavior of frameworks and components to be selected and changed flexibly. For example, the same interface can be used to obtain different types of loads on a given set of resources. Only object implementations must change since load measuring techniques may differ for each type of load. Each implementation is called a "strategy" and can be embodied in an object called a *load monitor*.

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

The Mediator [25] design pattern defines an object that encapsulates how objects will interact. In addition to playing the role of a strategy, a load monitor 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 to 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 5, the load balancer

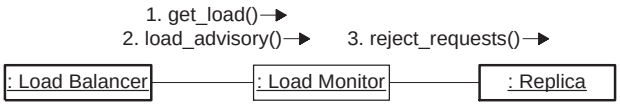

Figure 5: Feedback and Control when Balancing Loads

queries the load monitor for the load on the current replica, assuming that pull-based load monitoring is being used (see Section 3.1). In other words, the load balancer receives *feedback* from the load monitor. Load balancing control messages–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.

These two state values are the defaults provided by TAO. Users can define their own customized load states, however, by customizing the load analyzer and load monitor component implementations.

TAO's load balancer is *adaptive* due to the bi-directional feedback/control channel between the load monitor and the load balancer, which allows TAO's load balancer to administer *control*. Since the load monitor is decoupled from the load balancer it is also possible to balance loads across replicas based on various types of load metrics. For instance, one type of load monitor could report CPU loads, whereas another could report I/O resource load. 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.

4.0.3 Challenge 3: Supporting Modular Load Balancing Strategies

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

Problem: Since certain load analysis techniques are not suitable for all use-cases, it may be useful to analyze a set of 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, *e.g.*, if the duration of the analysis exceeds the time required to complete client request processing.

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

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

Solution $\rightarrow$ the Component Configurator pattern: The *Component Configurator* design pattern [22] 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 this pattern to adapt to different load balancing use-cases, without being hard-coded to handle just 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 situations, such as during periods of peak activity during the workday, a load balancing strategy may need modifications to account for increased load. In such cases, a more complex strategy may be necessary. The Component Configurator pattern makes it easy to dynamically configure load balancing algorithms appropriate for different use-cases *without* stopping and restarting the load balancer.

Applying the solution in TAO: TAO's load analyzer uses the Component Configurator pattern to customize the load balancing algorithm used when making load balancing decisions, as depicted in Figure 6. 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

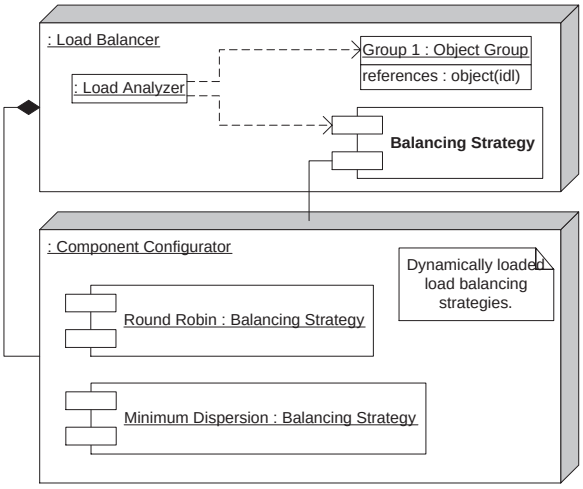

Figure 6: Applying the Component Configurator Pattern to TAO's Load Balancing Service

causes a request to be forwarded to the next replica in the object group being load balanced [18].

• **Minimum dispersion:** This adaptive strategy is more sophisticated than the round-robin algorithm described above. The goal of this strategy is to ensure load differences fall within a certain tolerance, *i.e.*, it attempts to ensure that the average difference in load between each replica is minimized. The following steps are used in this on-demand adaptive strategy:

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.

A large amount of work on load balancing strategies [26] has already been done. Many of those same strategies can be integrated in to the CORBA-based load balancing service via the Component Configurator pattern implementation described above.

4.0.4 Challenge 4: Coping With Adaptive Load Balancing Hazards

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

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

• **Thundering herd:** When a less loaded replica suddenly becomes available, a "thundering herd" phenomenon may occur if the load balancer forwards all requests to that replica immediately. 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 effectively. 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 4.0.3 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 less 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.

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 [30] illustrate the effects of TAO's dampening mechanisms.

4.0.5 Challenge 5: 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 4.0.2. 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.[7] Moreover, as described in Section 3.1, CORBA provides *weak identity* for objects, relying on the replica object reference to distinguish them would not be 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 [22]. 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 7. The load balancing service creates a unique replica proxy for each monitor. When the replica proxy implementation 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 `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.

[7] The CORBA Security Service [27] can authenticate client requests, but this is a much more expensive mechanism than required for many applications.

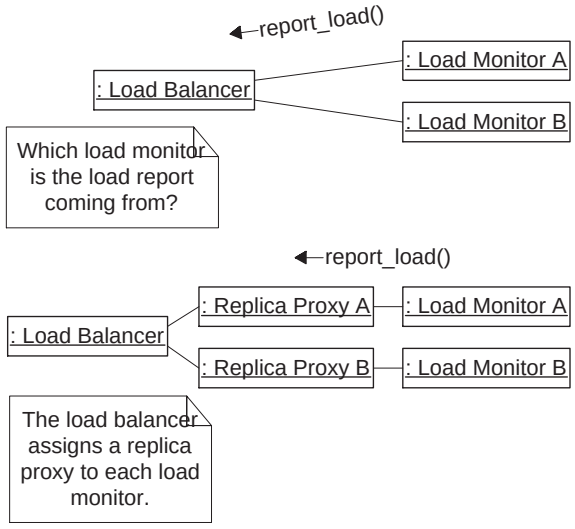

Figure 7: Identifying the Source of a Message Uniquely

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.

4.0.6 Challenge 6: Integrating All the Load Balancing Components Effectively

Context: As illustrated above, a load balanced distributed system has many components that interact with each other. For example, clients issue requests to 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, however, since certain functionality may be exposed to a given component unnecessarily.

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

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

A *load balancer* component can be used to tie together all the components listed above. It coordinates all interactions between other components, *i.e.*, it is a mediator. 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 1, 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 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 these loads were obtained is hidden from the load analyzer.

5 Concluding Remarks and Future Work

This paper describes the design of an adaptive middleware-based load balancing service developed for the TAO ORB [3]. 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. TAO's load balancing service is based entirely on standard features in CORBA. This implementation demonstrates that CORBA technology has matured to the point where many higher-level services can be developed effectively 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 poorly understood features, such as location forwarding. We believe that further research on effective architectures, strategies, and patterns to implement CORBA load balancing services is necessary to advance the state of the art. Below, we outline future work that we are conducting to improve our CORBA load balancing service.

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 [28, 24] can capture client requests transparently before they are dispatched to an object 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 modification to server application code is necessary, other than registering the interceptor with the ORB when it starts running.

• **CORBA Component Model (CCM):** The CCM [21] introduces *containers* to decouple application component logic from the configuration, initialization, and administration of servers. In the CCM, a container creates the POA and interceptors required to activate and control a component. These are the same CORBA mechanisms used to implement the server components in TAO's load balancing service. The standard CCM containers can be extended to implement automatic load balancing *generically* without changing application component behavior.

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. This model simplifies the design

and implementation of the load balancer, but introduces a single point of failure, which can impede system reliability and scalability.

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 we will address 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. Techniques used 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 many objects in the server then instantiating a load monitor (see Section 4.0.2) for each object may not scale. For example, load monitor resources, such as memory, CPU, and 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 a more scalable load monitoring granularity. 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.

We believe this design can significantly reduce the amount of resources imposed by adding 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 a load balancer detects a high load and issues a load advisory to the shared load monitor. The load monitor must now 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.

Other problems can occur when multiple object groups reside on a single server. Load balancing decisions for one object group may actually interfere with load balancing decisions for another object group. Suppose both object groups are balanced based on CPU load. The load balancer detects low load conditions for the first object group, causing requests to be sent to that object group, which causes the CPU load to increase on the given server. Since the second object group is load balanced based on CPU load, the load balancer will detect a high load on the server due to the increased load caused by the requests sent to the first object group. At this point, the load balancer will cause the second object group to reject requests. Thus, the second object group is starved by the first object group. In this scenario, the two object groups must be load balanced collectively, which implies a common load monitor must be used for both object groups.

Fault tolerant load balancing: By using the adaptive CORBA-based load balancing architecture described in this paper, clients that have not been forwarded to replicas can still be denied service. Some form of fault tolerance is therefore 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 Fault Tolerant CORBA can alleviate one of the inherent problems with centralized load balancing: its single point of failure. It can also ensure that state within 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 [10, 29]. Since DOORS itself is a CORBA service implemented using TAO integrating it with TAO's load balancer should be straightforward.

Improved quality of service support: As mentioned in Section 4.0.4, 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.

Some approaches that can be used to improve the effectiveness of a given load balancing strategy are:

- 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 it is perturbed by external forces. In the case of load balancing, external forces could be additional client requests or transient loads generated by other applications running over the network and end-systems.

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

Advanced replica management: It is common practice to design a service that balances loads across a group of replicas supplied to it by applications explicitly. In particular, TAO's load balancing service described in this paper makes no attempt to control replica lifetime. More advanced solutions, however, can determine how replicas are created and destroyed.

For example, suppose there are only two replicas in a replica group and that their loads are high. Without additional replicas, it may be hard to maintain balanced loads. A load balancing service with the ability to create and destroy replicas on-demand may provide more flexible load balancing strategizes, *e.g.*, a load balancer could create a replica at a third location in an effort to decrease the workload on the two initial replicas.

Those familiar with fault tolerance services may recognize a similarity between their replica management strategies and those of load balancing services. Both types of services can control replica lifetimes, *e.g.*, by creating replicas on-demand. A fault tolerance service requires sufficient replicas to provide fault recovery, while a load balancing service requires enough replicas to provide balanced loads. Although the underlying functionality for each type of service is different, the interface exposed by each service can be similar. Therefore, the IDL interfaces exposed by TAO's next-generation load balancing service under development currently is based largely on the IDL interfaces standardized by the Fault Tolerant CORBA specification [20].

TAO and TAO's load balancing service have been applied to a wide range of distributed applications, including many telecommunication systems, aerospace/military systems, online trading systems, medical systems, and manufacturing process control systems. All the source code, examples, and documentation for TAO, its load balancing service, and its other CORBA services is freely available from URL `http://www.cs.wustl.edu/~schmidt/TAO.html`. A paper describing the performance of TAO's load balancing service appeared in [30].

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

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

[3] 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.

[4] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, To appear 2001.

[5] 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.

[6] 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.

[7] 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.

[8] 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 2001.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] O. Othman, C. O'Ryan, and D. C. Schmidt, "The Design of an Adaptive CORBA Load Balancing Service," *IEEE Distributed Systems Online*, vol. 2, Apr. 2001.

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

[16] 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.

[17] 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.

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

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

[22] 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.

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

[24] 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.

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

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

[27] Object Management Group, *Security Service 1.8 Specification*, OMG Document security/00-11-03 ed., November 2000.

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

[29] 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.

[30] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.