Optimizing Distributed System Performance via Adaptive Middleware Load Balancing

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

Abstract

Load balancing middleware is used extensively to improve scalability and overall system throughput in distributed systems. Many load balancing middleware services are simplistic, however, since they are geared only for specific use-cases and environments. These limitations make it hard to use the same load balancing service for anything other than the distributed application it was designed for originally. This lack of generality forces continuous re-development of application-specific load balancing services. Not only does re-development increase deployment costs of distributed applications, but it also increases the potential of producing non-optimal load balancing implementations since proven load balancing service optimizations cannot be reused directly.

This paper presents a set of load balancing service features that address many existing middleware load balancing service inadequacies, such as lack of server-side transparency, centralized load balancing, sole support for stateless replication, fixed load monitoring granularities, lack of fault tolerant load balancing, non-extensible load balancing algorithms, and simplistic replica management. All the capabilities described in this paper are currently under development for the next generation of middleware-based load balancing service distributed with our CORBA-compliant ORB (TAO).

Keywords: Middleware, patterns, scalability, CORBA, load balancing.

1 Introduction

Motivation: As the demands of resource-intensive distributed applications have grown, the need for improved overall throughput and scalability has also grown. A cost-effective way to address these application demands is to employ load balancing services based on distributed object computing *middleware*, such as CORBA [1] or Java RMI [2]. These load balancing services distribute client workload equitably among various back-end servers in order obtain the best response time possible given a particular load.

Many existing middleware load balancing services provide just enough functionality to support simple distributed applications. For example, stateless distributed applications that require load balancing often employ a simple load balancing service that is integrated with a naming service [3, 4]. In this approach, a naming service returns a reference to a different object each time it is accessed by a client. Load balancing via a naming service only supports a static non-adaptive form of load balancing, however, which limits its applicability to distributed systems with more sophisticated load balancing needs. This approach also greatly reduces the potential for optimizing overall distributed system load since it is not possible to adapt the behavior of the load balanced applications dynamically.

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

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

However, first-generation adaptive middleware load balancing services [5, 6, 7, 8] do not provide solutions for key dimensions of the problem space. In particular, they provide insufficient functionality to satisfy complex distributed applications with higher optimization requirements. In general, as the complexity of distributed applications grows, their load balancing requirements necessitate more advanced functionality, such as

[*]This work was funded in part by ATD, BBN, Cisco, DARPA contract 9701516, and Siemens MED.

the ability to tolerate faults, install new load balancing algorithms at run-time, and create replicas on-demand to handle bursty clients. The lack of this advanced functionality can adversely effect distributed system performance and scalability. This paper discusses these and other types of load balancing functionality necessary to optimize complex distributed systems more effectively.

Background: This paper assumes that readers are familiar with basic load balancing concepts and terminology. The key conceptual components used in this paper are outlined below:

- **Load balancer:** This component attempts to ensure that loads are balanced across a group of servers. It is sometimes referred to as a "load balancing agent," or a "load balancing service." In this paper, a load balancer may consist of a single centralized server or multiple *decentralized* servers that collectively form a single logical load balancer.

- **Replica:** This component is a duplicate instance of a particular object on a server that is managed by a load balancer. They can either retain state (*i.e.*, be *stateful*) or retain no state at all (*i.e.*, be *stateless*).

- **Replica group:** This component is actually a group of replicas across which loads are balanced. Replicas in such groups implement the same remote operations. Replica groups are also referred to as "object groups."

- **CORBA and TAO:** 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 [9]. TAO is an open-source[1] CORBA-compliant ORB designed to address applications with stringent quality of service (QoS) requirements.

The structure and relationship of these components is illustrated in Figure 1. The TAO CORBA ORB provides the distribution middleware for all of the components shown in this figure. TAO facilitates location-transparent communication between:

- Clients and a load balancer;
- A load balancer and the replicas; and
- Clients and the replicas

In this case, the load balancer also keeps track of which replicas belong to each replica group. For more details on load balancing concepts and terminology see [5].

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

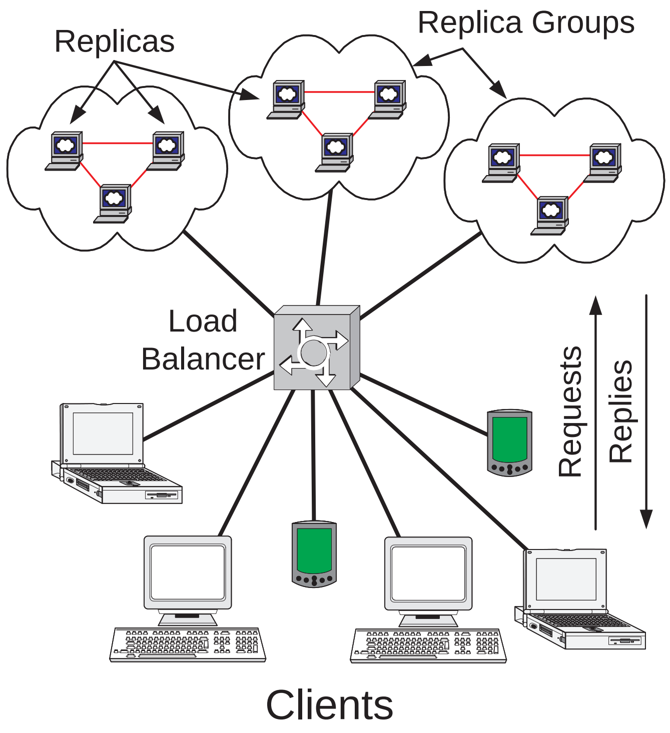

Figure 1: CORBA-based Load Balancing Concepts and Components

Paper organization: The remainder of this paper is organized as follows: Section 2 summarizes a set of advanced CORBA-based[2] load balancing features designed to address optimization and inadequacies in existing load balancing services; Section 3 briefly describes related load balancing work; and Section 4 presents concluding remarks.

2 Optimizing and Enhancing Load Balancing Services

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

1. Server transparency
2. Decentralized load balancing
3. Stateful replicas

[2]Although this paper concentrates primarily on CORBA-based load balancing services, the concepts outlined in this paper are applicable to load balancing services that are based on other types of middleware, such as COM+, Java RMI, and proprietary message-oriented middleware (MOM).

4. Diverse load monitoring granularity
5. Fault tolerant load balancing
6. Extensible load balancing algorithms
7. On-demand replica activation

We explore each of these features below.

2.1 Server Transparency

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

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

Solution → the Component Configurator and Interceptor patterns: If adaptive load balancing is to be used transparently on the server-side of a distributed application, there must be some way to install feedback/control mechanisms into the server without altering the server application software. Fortunately, most ORB middleware–and in particular CORBA–provide a meta-programming mechanism based on the Interceptor pattern [10]. These mechanisms can alter the behavior of a client or a server when processing a given client request [11]. An interceptor can be installed at run-time to provide the functionality necessary to (1) communicate with the load balancing service and (2) accept load control requests from the load balancing service. Since the interceptor mechanism is part of the middleware implementation, server application software need not be modified.

To provide true server-side transparency, however, there must be some means of installing interceptors transparently to control requests from the adaptive load balancing service. The Component Configurator pattern [10] can be used to dynamically load a service into an application at run-time. In particular, a Component Configurator can be used to transparently install a load balancing interceptor into an application's underlying middleware at run-time, as illustrated in Figure 2. Using this approach, the overall throughput of a distributed application can be improved without modifications to distributed application server code.

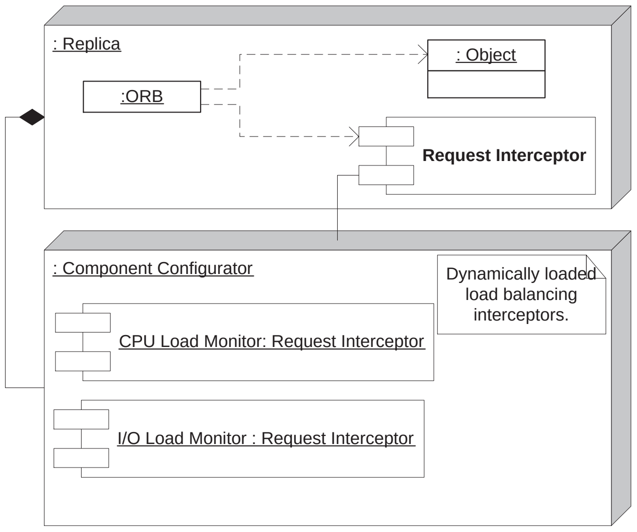
: Replica
:ORB
: Object
Request Interceptor
: Component Configurator
Dynamically loaded load balancing interceptors.
CPU Load Monitor: Request Interceptor
I/O Load Monitor : Request Interceptor

Figure 2: Transparent Server-side Load Balancing

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

- **Portable interceptors:** Portable interceptors [12] 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 [13] 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.

2.2 Decentralized Load Balancing

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

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

Solution → Federated architecture: To overcome the problems in centralized load balancing services, a *federated* load balancing architecture can be used to implement a more scalable and reliable load balancing service. In this model, load balancing is performed via a distributed, *i.e.*, decentralized, set of load balancers that collectively form a single *logical* load balancing service. This architecture is illustrated in Figure 3.

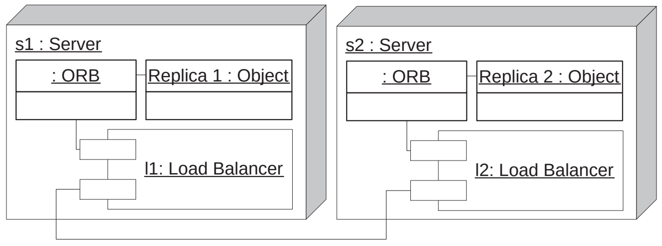

Figure 3: Federated Load Balancing

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

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

Applying the solution in TAO: The techniques described in Section 2.1 to ensure server-side load balancing transparency can be used to implement a federated load balancing service. In particular, the "distributed" component of a federated load balancing service can reside in an interceptor that is installed transparently.

TAO's next-generation adaptive load balancer uses the Component Configurator pattern to dynamically load a factory object. That factory object creates an ORB initializer [12] object that registers the load balancing interceptor with an ORB each time an ORB is created. This design allows a portion of a federated load balancing service to transparently reside at multiple locations, *i.e.*, where ever replicas reside.

2.3 Stateful Replicas

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

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

Solution → the Memento pattern: To load balance replicas that retain state, some means of maintaining state consistency between replicas is necessary. The Memento pattern [14] can help to address this need by capturing internal state so that it may be restored at a later time. For example, a load balancing service could invoke `get_state` and `set_state` operations on a pair of replicas to transfer state between the replicas. These two methods are specified by the Memento pattern, and the replica itself must implement them. Figure 4 illustrates the sequence of operations that occur when forwarding a client to a less load stateful replica. These operations are outlined below:

1. A client makes a request. The request is intercepted by the load balancer transparently.
2. To transfer load to a new replica, the load balancer obtains the state from the overloaded replica using the `get_state` operation.
3. That state is then restored into the new underloaded replica by invoking `set_state` operation on that underloaded replica.
4. After the state transfer occurs, the new replica can service client requests, and the load balancer notifies it that it can begin accepting requests.

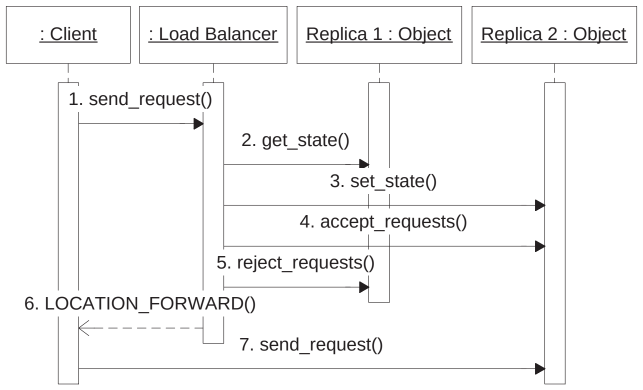

Figure 4: Load Balancing Stateful Replicas

5. The overloaded replica must shed some of its load, so the load balancer notifies it that it should reject requests. This entails making the overloaded replica redirect client requests back to the load balancer.
6. The load balancer now redirects the client to the new underloaded replica transparently by means of the GIOP LOCATION_FORWARD message.
7. The client ORB reissues the request to the new less loaded replica.

An alternative solution is provided by the CORBA Persistent State Service [15]. It extends the CORBA IDL[3] so that it becomes possible for distributed application developers to define precisely what the internal state is, *i.e.*, its format or schema. This *a priori* knowledge facilitates persistence at compile-time, and thus simplifies the automation and transfer of state in distributed applications.

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

2.4 Diverse Load Monitoring Granularity

Context: A server has multiple objects running on it, each of which must be load balanced. Moreover, multiple servers may be running at a single location, and these servers must also be load balanced. To accomplish this, a load balancer requires a load monitoring object from which it can obtain the current load conditions wherever one or more replicas reside.

Problem: As with any object, an instance of a load monitoring object utilizes resources. Instantiating a load monitoring object for each replica may not scale if there are many replicas in a server. For example, allocating load monitoring resources (such as memory, CPU, and network bandwidth) for each load monitoring object can starve other objects or processes running on a server since the load monitoring objects themselves impose their own resource utilization overhead.

Other problems can occur when multiple replica groups reside on a single server. Load balancing decisions for one replica group may interfere with load balancing decisions for another replica group. For example, consider the case where two replica groups are balanced based on CPU load. The load balancer detects low load conditions for the first replica group, causing requests to be sent to that replica group, which causes the CPU load to increase on the given server. Since the second replica 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 replica group. At this point, the load balancer will cause the second replica group to reject requests. Thus, the second replica group is starved by the first replica group.

Solution $\rightarrow$ shared load monitors: Rather than instantiating a load monitor for each object on the server, a single load monitor can 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. Figure 5 depicts this approach.

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 complicates the load monitor implementation. For example, suppose a load balancer detects a high load and issues a load advisory[4] 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.

This approach can be generalized by creating a load monitor object hierarchy to further reduce the number of messages required to communicate load information to the load balancing service, thus reducing network bandwidth requirements. However, this approach can complicate the load balancing service and load monitor implementations. In general, improving per-

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

[4]A load advisory is a message sent to a load monitor object that lets it know the relative load conditions at its location.

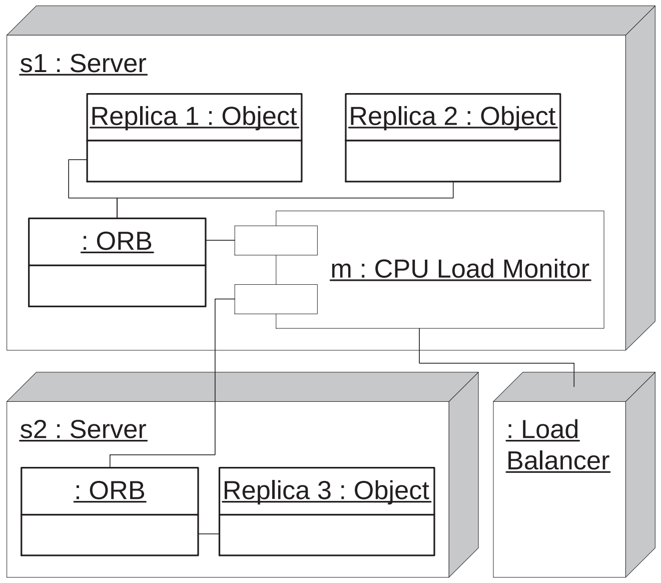

Figure 5: Shared Load Monitor

formance increases load balancing service complexity to some extent.

2.5 Fault Tolerant Load Balancing

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

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

Solution → a fault tolerance service: Since CORBA-based load balancing services are themselves CORBA applications, the standard CORBA Fault Tolerance service [16] can be used to provide the means by which a load balancing service remains highly available. Making a load balancing service fault tolerant by means of Fault Tolerant CORBA can alleviate one of the inherent problems with centralized load balancing: its single point of failure. It can also ensure that state within 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 [17, 18]. Since DOORS itself is a CORBA service implemented using TAO, integrating it with TAO's load balancer should be straightforward, for example.

2.6 Extensible Load Balancing Algorithms

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

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

Solution → the Component Configurator and Strategy patterns: As stated in Section 2.1, the Component Configurator pattern defines a means by which a component can be loaded dynamically into a running application. By taking advantage of the extensibility provided by an implementation of this design pattern, customized load balancing algorithms can be configured into a running load balancing service. Specifically, load balancing algorithms employed by the load balancing service can be implemented via the Strategy pattern [14]. The combination of these two patterns allows a load balancing service to cope with degenerate load conditions, in addition to further generalizing the applicability of the load balancing service to other types of distributed applications. Figure 6 illustrates how this solution is deployed.

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

- Take into account past load trends when predicting future load conditions.
- Take advantage of sophisticated algorithms [19] that are designed specifically to restore system equilibrium when it is perturbed by external forces. In the case of load balancing, external forces could be additional client requests or transient loads generated by other applications running over the network and end-systems.
- Make load balancing decisions based on multiple load metrics, which requires the ability to send multiple loads

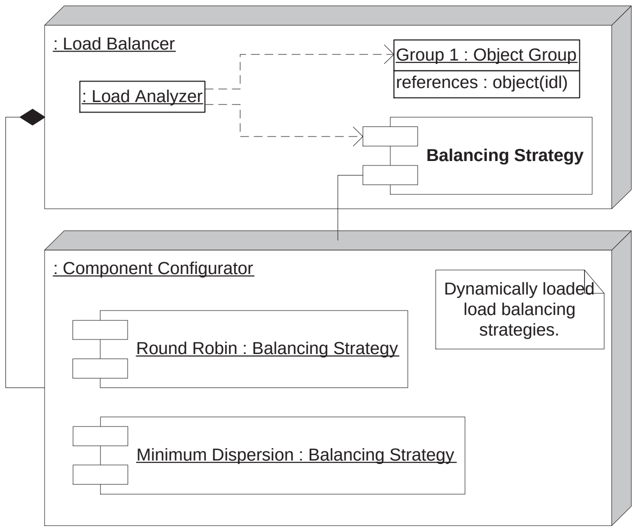

Figure 6: Extensible Load Balancing Algorithms

in a single load report. For example, a load balancing strategy could receive a sequence of load metrics that correspond to multiple load readings, each of a different type, at a given location. The CORBA IDL for such a sequence of loads could be the following:

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

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

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

2.7 On-demand Replica Activation

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

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

Solution $\rightarrow$ the Factory pattern: The Factory pattern [14] exposes an interface through which objects can be created. A load balancing service can use factory objects, *i.e.*, objects that implement the Factory design pattern, to create replicas on-demand. The load balancing service would simply invoke remote operations on the factory object at a given location when it decides that more replicas are necessary to maintain balanced loads.

For example, suppose there are only two replicas in a replica group and that their loads are high. Without additional replicas, it may not be feasible to maintain balanced loads. A load balancing service with the ability to create and destroy replicas on-demand may provide more flexible load balancing strategies, *e.g.*, a load balancer can create a replica at a third location to decrease the workload on the two initial replicas, as shown in Figure 7.

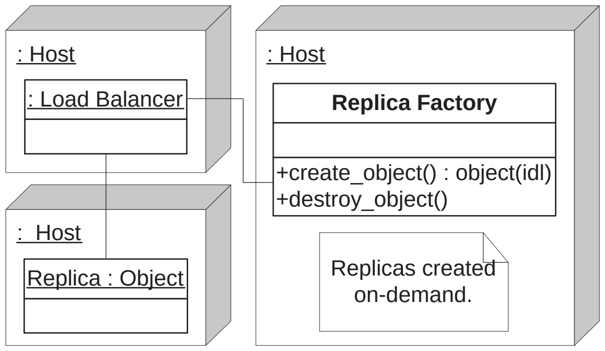

Figure 7: On-demand Replica Creation

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

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, currently under development, is based largely on the IDL interfaces standardized by the Fault Tolerant CORBA specification [16].

3 Related Work

Extensive research on load balancing algorithms [19] has been done. This paper concentrates on mostly on load balancing service functionality rather than the underlying load balancing algorithms. A significant amount of work has been done on load balancing services at various system levels. This includes research on load balancing services at the network, the operating system, and the middleware, as described below.

Network-based load balancing: Network-based load balancing services make decisions based on the frequency at which a given site receives requests [20]. For example, routers [21] and DNS servers often perform network-based load balancing. Load balancing performed at the network level has the disadvantage that load balancing decisions are based solely on the destination of the request. The content of the request is often ignored. This form of load balancing also makes it difficult to select the load metric to be used when making balancing decisions.

OS-based load balancing: Load balancing at the operating system level [22, 23, 24] has the advantage of performing the balancing at multiple levels. That balancing is essentially transparent to a distributed application. However, it suffers from many of the same problems that network-based load balancing suffers from, such as inflexible load metric selection and not being able to take advantage of request content. OS-based load balancing may also be too coarse-grained for some distributed applications where it is the objects residing within a server, rather than the server process itself, that must be load balanced.

Middleware-based load balancing: Middleware-based load balancing provides the most flexibility in terms of influencing how a load balancing service makes decisions, and in terms of applicability to different types of distributed applications [25]. Load balancing at this level provides for straightforward selection of load metrics, in addition to the ability to make load balancing decisions based on the content of a request. Some middleware-based implementations integrate load balancing functionality into the ORB middleware [7] itself, whereas others implement load balancing support at the service level. The latter is the approach taken by the TAO next-generation load balancing service upon which the content of this paper was based.

4 Concluding Remarks

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

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

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

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

TAO's next-generation load balancer will support the functionality outlined above. Transparent server-side load balancing for stateless replicas will be supported by the standard CORBA portable interceptors [12] mechanism. Federated load balancing will be implemented via reliable multicast. State migration will be supported by using the CORBA Persistent State Service [15] being developed for TAO. Different load monitoring granularity levels will be supported via the CORBA portable interceptor mechanism, in addition to hierarchical load monitoring. Basic fault tolerance will be supported through CORBA Fault Tolerance service implementation [17, 18] currently being developed for TAO. Extensible load balancing strategies are already supported by TAO. Finally, TAO's run-time control of replica life times will capitalize on the interface provided by the CORBA Fault Tolerance specification.

References

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

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

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

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] W. G. Krebs, "Queue Load Balancing / Distributed Batch Processecing and Local RSH Replacement System." http://www.gnuqueue.org/home.html, 1998.

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