IEEE distributed systems ONLINE
Expert-authored articles and resources

*IEEE DS Online*, Volume 2, Number 4

# Designing an Adaptive CORBA Load Balancing Service Using TAO

**Ossama Othman, Carlos O'Ryan, and Douglas C. Schmidt**
University of California, Irvine

The growth of online Internet services during the past decade has increased the demand for scalable and dependable distributed computing systems. These systems face high quality-of-service requirements (such as immediate response time and 24/7 availability) and concurrently serve many clients that transmit a large, often bursty, number of requests. E-commerce and online stock trading systems are examples.

To protect hardware investments and avoid overcomitting resources, such systems scale incrementally by connecting servers through high-speed networks. Load balancing helps improve system scalability by distributing and processing client application requests equitably across a group of servers. Likewise, it helps improve system dependability by adapting dynamically to system configuration changes that arise from hardware or software failures. Load balancing also helps ensure all resources are used efficiently and that new servers are purchased or cycles are leased by companies only when necessary.

In "Strategies for CORBA Middleware-Based Load Balancing" (DS Online, March 2001), we discussed performance issues and load balancing strategies and introduced a CORBA load balancing service we designed using TAO—the ACE (Adaptive Communication Environment) ORB.



IEEE Distributed Systems Online Published by the IEEE Computer Society 1541-4922/01/$17.00 @ 2001 IEEE

Building on Part I, this article (Part II) focuses on the specifications for middleware-based load balancing mechanisms developed using standard CORBA, and discusses the specific load balancing service we designed using TAO.

## Requirements for CORBA load balancing services

CORBA's rich set of features and inherently distributed nature make it ideal for implementing load balancing in distributed systems. CORBA has a common heterogeneity of clients and servers written in different programming languages running on different hardware and software platforms. A CORBA load balancing service can take full advantage of the request forwarding mechanism the CORBA specification mandates.[1] A CORBA server application can use this mechanism to forward client requests to other servers transparently, portably, and interoperably. Thus, CORBA simplifies system implementation by providing a language- and platform-neutral communication infrastructure. It also reduces development effort by offering higher level programming abstractions that shield application developers from distribution complexities, thereby letting them concentrate on the application instead of managing system resources.

The Object Management Group's CORBA specification provides the core capabilities needed to support load balancing. The design of an effective CORBA load balancing service should be based on the following requirements:

> Support an object-oriented load balancing model: In the CORBA programming model, objects are the unit of abstraction, and system architects reason about objects to manage their available resources. Thus, the granularity of load balancing in CORBA should be based on objects rather than processes or TCP/IP addresses, for example. Moreover, a load balancing service and ORB should coordinate the interactions among multiple object replicas. Sets of multiple object replicas are called object groups or replica groups.
>
> Enable client application transparency: Distributing workload

among multiple servers should require little or no modification to normal CORBA application development. 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 should therefore require no changes to clients whose requests it balances.

Enable server application transparency: Although load balancing should ideally require few modifications to servers, this is difficult to implement. For example, load balancing a stateful CORBA object requires transferring its state to a new replica. The application implementation must either perform the transfer itself or define hooks that let the load balancing framework perform the state transfer as unobtrusively as possible.[2] The situation for stateless CORBA servers is different. In this case, implementing a server object's servant (a programming language entity that implements object functionality in a server application) should require no changes to support load balancing. Yet changes to the server application might still be required under certain conditions. For example, some applications might define ad hoc load metrics such as the number of active transactions or user sessions. In practice, collecting these metrics might require some modifications to server application code.

Support dynamic client operation request patterns: We can base load balancing services on various client request patterns. For example, load balancers for certain types of systems assume client requests occur at deterministic or stochastic rates executing for known or fixed durations of time. While these assumptions might apply for certain types of applications, such as continuous multimedia streaming,[3] they do not apply in complex Internet or military[4] environments where client operation request patterns are dynamic and the duration of each request might not be known in advance. In this article, 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 (for example, based on extensions to naming services), these approaches do not balance dynamic loads equitably and their scalability is limited. (IONA Technologies' Orbix 2000 http://www.iona.ie/products/orbix2000_home.htm is an example of a naming-service based load balancer.) Thus, a CORBA load balancing service must increase system scalability by maximizing dynamic resource utilization in a group of servers that otherwise would be underutilized. By improving resource utilization via load balancing, the overall scalability of the server group should improve 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. However, load balancing services need not provide full fault-tolerance capabilities (it should not be the role of a load balancing service to detect and mask failures).[5,6] Instead, they should provide mechanisms to handle those failures efficiently when detected by administrators or other system components.

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

Incur minimal overhead: A CORBA load balancing service should not introduce undue latency or networking overhead because to do so reduces—rather than enhances—overall system performance. In particular, an implementation that increases the average number of messages per-request or uses a single server to process all requests might be inappropriate for high-performance or large-scale applications.

Support application-defined load metrics and balancing policies: Different types of applications have different notions of load. This support can be targeted solely to the server replica, without affecting client transparency. Thus, a CORBA load balancing service should let applications specify the semantics of metrics used to measure load. For example, some applications will want to balance CPU load; others will be more concerned with balancing I/O resources, communication bandwidth, or memory load. It should also set policies that determine the load balancing service's semantics. For example, some applications will want to distribute load uniformly, others randomly, and still others will want load distributed based on dynamic metrics (such as current CPU load or current time).

Rely on 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, the standard protocols enabling heterogeneous CORBA clients and servers to interoperate. Likewise, avoid implementing load balanced objects by adding proprietary extensions to an ORB.

## The design of a TAO load balancing service for CORBA

TAO—the ACE (Adaptive Communication Environment) ORB is a CORBA-compliant ORB that supports applications with stringent QoS requirements.[7] Our design for TAO's load balancing service makes it easier to develop distributed applications in heterogeneous environments by providing application transparency, high flexibility, scalability, runtime adaptability, and interoperability.

### Component structure in TAO's load balancing service

Figure 1 illustrates the components in TAO's load balancing service. (The term component used throughout this article refers to a "component" in the general sense—that is, an identifiable entity in a program rather than in the more specific sense of the CORBA

Component Model.) Our design supports adaptive load balancing and on-demand request forwarding, as outlined below:

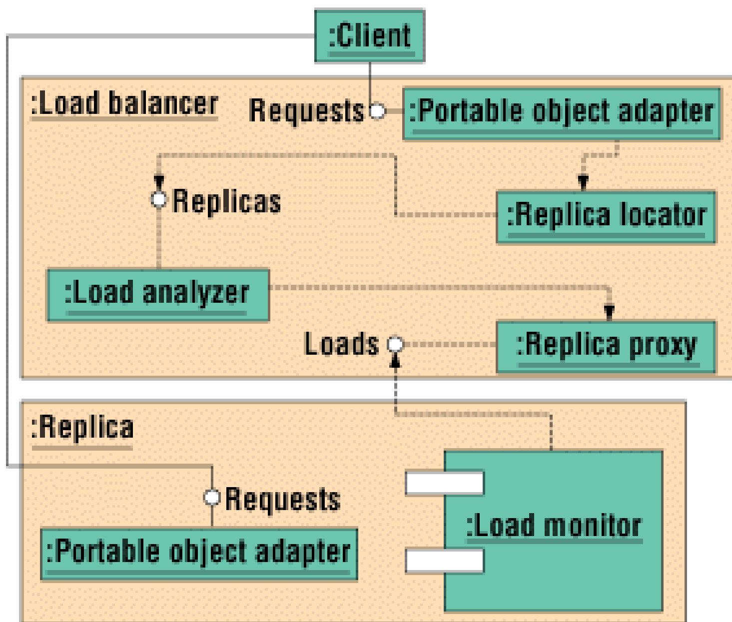



**Figure 1. Components in the TAO load balancing service.**

Replica locator: This component identifies which replicas will receive which requests. It also binds clients to the identified replicas. The replica locator can be implemented portably using standard CORBA portable object adapter mechanisms (such as servant locators[8] implementing the interceptor pattern).[9] The replica locator forwards each request it receives to the replica the load analyzer selects.

Load monitor: This component monitors loads on a given replica, reports replica load to a load balancer, and responds to load advisories the load balancer sends. A load monitor also processes load advisories the load balancer sends and informs replicas when they should accept requests versus forward them back to the load balancer. As Figure 2 shows, a load monitor can be configured with either a pull or push policy. In the pull mode, a load balancer can query a given replica load on-demand; it "pulls" loads from the load monitor. In the push mode, a load monitor can "push" load reports to the load balancer.

Load analyzer: This component decides which replica will receive

the next client request. The replica locator obtains a reference to a replica from the load analyzer and then forwards the request to that replica. The load analyzer also allows explicit selection of a load balancing strategy at runtime, while maintaining a simple and flexible design. Because it is possible to choose the load balancing strategy at runtime, the application developer can tailor replica selection to fit the dynamics of a system being load balanced.

Replica proxy: Each object that TAO's load balancing service manages communicates with it through 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,[6] where two references to the same object might 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. If they don't refer to the same object, either they're not equivalent or the ORB couldn't make this determination.

Load balancer: This component is a mediator that integrates all the components we just described. It provides an interface for load balancing without exposing clients to the intricate interactions between the components it integrates.

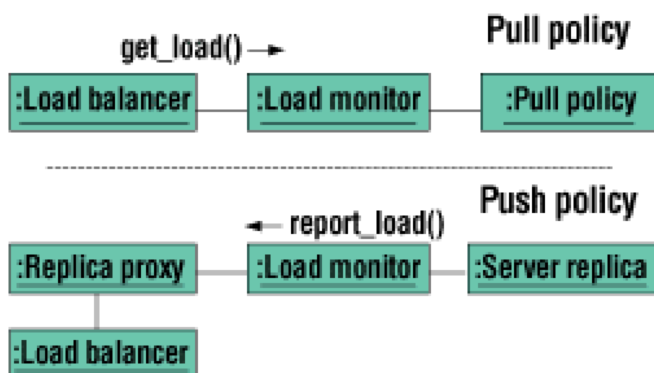
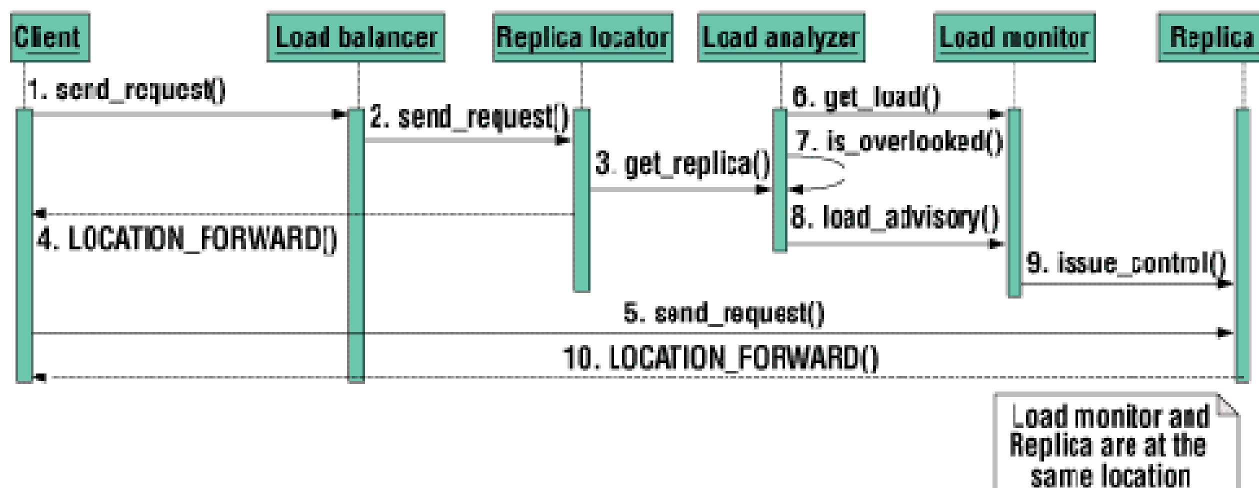



**Figure 2. Load reporting policies.**

## Dynamic interactions in TAO's load balancing service

As described in Part I of this article, selecting a target replica using a nonadaptive balancing policy can yield nonuniform loads across replicas. In contrast, selecting a replica adaptively for each request can incur excessive overhead and latency. Therefore, to avoid either extreme, TAO's load balancing service provides a hybrid solution through one of its load balancing strategies. Figure 3 shows the interactions.




1. A client obtains an object reference to what appears to be a replica and invokes an operation. Actually, the client transparently invokes the request on the load balancer itself.
2. After receiving the request from the client, the load balancer's Portable Object Adapter dispatches the request to its servant locator—that is, 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. Clients will continue to send requests directly to the chosen replica until the load balancer detects a high load on that replica. The client's direct communication with the replica eliminates the additional overhead incurred by per-request load balancing architectures.
6. The load balancer monitors the replica's load. Depending on the load reporting policy 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 the load balancer collects loads, 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 transparency requirements, 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 advisor to the replica's load monitor.
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.

**Figure 3. TAO load balancer interactions.**

# Design challenges and their solutions

We faced several design challenges before and during development of TAO's load balancing service. We had to:

1. implement portable load balancing,
2. enhance feedback and control,
3. support modular load balancing strategies,
4. cope with adaptive load balancing hazards,
5. identify objects uniquely, and
6. integrate all of the load balancing components effectively.

The challenges are presented here in pattern form using the headings Context, Problem, Solution, and Applying the solution to TAO to show a concrete example on how the generic pattern (or pattern-based solution) applies to one system. Pattern form is a technique developed to describe patterns—that is, recurring solutions to common problems. The pattern form technique not only describes the solution, but also describes the problem within the context of the general environment where the problem must be
solved. The real insight of patterns, and what has made them so successful, is the realization that solutions only make sense when they are solving the same problem in the same context.

## Challenge 1: Implementing portable load balancing

Context: A CORBA load balancing service is implemented in accordance with the requirements presented earlier.

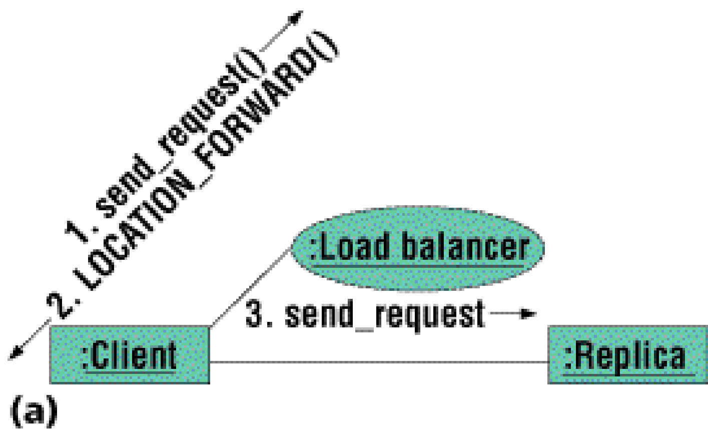
Problem: Changing application code—particularly client applications—to support load balancing can be tedious, error-prone, and costly. Changing the middleware infrastructure to support load balancing is

also problematic, because the same middleware may be used in applications that do not require load balancing (and the extra overhead and footprint might be unacceptable). Likewise, using ad hoc or proprietary interfaces to add load balancing to existing middleware can increase required maintenance and might 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[9] lets a framework 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. A dispatcher invokes methods in this interface.

The Interceptor pattern can be implemented via standard CORBA POA[1] features. For example, a servant locator plays the role of the interceptor and the POA plays the role of the dispatcher. Servant locators are a metaprogramming mechanism[11] enabling CORBA server application developers to obtain custom object implementations dynamically, rather than using the POA's active object map.[12] In particular, a replica locator can implement the standard CORBA servant_locator[1] interface the POA provides. Figure 4 illustrates how loads are balanced transparently using standard CORBA features.



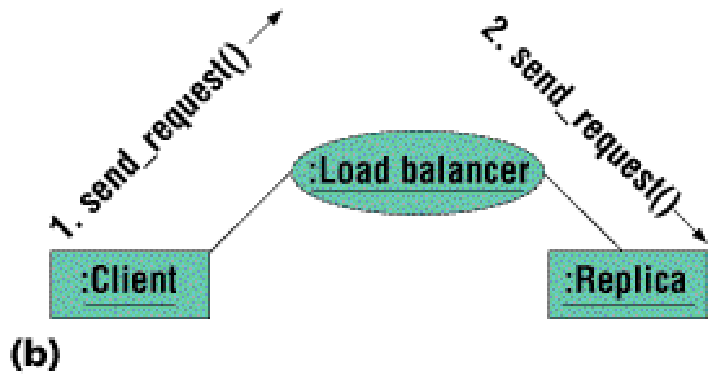




**Figure 4. Load balancing transparency in applications: (a) request forwarded by the client and (b) request forwarded on behalf of the client.**

Initially, clients use standard CORBA mechanisms to obtain 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 the application selects, either the client forwards requests to the appropriate replica, as Figure 4a shows, or the load balancer forwards requests to the appropriate replica on behalf of the client, as Figure 4b shows.

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 the load balancer intercepts. A servant locator performs the interception.

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—without affecting the server application code.

After receiving a request, the replica locator obtains a reference to the replica the load analyzer chooses and throws a forward_request 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. Reissue the request to the new location specified by the object references embedded in the location_forward response; and

2. 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.

## Challenge 2: Enhancing feedback and control

Context: An adaptive load balancing service must determine the current load conditions on replicas with which it is registered; however, a load balancer should not need to know the type of load metric beforehand. Moreover, a load balancer must take steps to ensure that loads across its registered replicas are balanced. These steps include forcing the replica to redirect the client back to the load balancer when its load is high and 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 is not transparent, a load balancer must sample loads from server replicas directly, which is undesirable because 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. 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 —> the Strategy and Mediator patterns: The Strategy design pattern[13] lets application developers and architects select and change the behavior of frameworks and components. 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 because 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, because one load balancer can balance many different types of load.

The Mediator design pattern[13] 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 the replica exports.

In its capacity as a mediator, a load monitor responds to load balancing requests the load balancer sends. 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 through 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 queries the load monitor for the load on the current replica, assuming that pull-based load monitoring is being used (discussed earlier). In other words, the load balancer receives feedback from the load monitor. The load balancer then sends load balancing control messages—called load advisories—to the load monitor, which 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.

**Figure 5. Feedback and control when balancing loads.**

These two state values are the defaults TAO provides. 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 bidirectional feedback and control channel between the load monitor and the load balancer; this lets TAO's load balancer administer control. Because 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. 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.

## Challenge 3: Supporting modular load balancing strategies

Context: A distributed system uses a load balancing service to improve overall throughput by ensuring 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: Because 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, some systems predict future replica loads, using the history of loads for a given object group and thus anticipate high load conditions. Conversely, this level of analysis is too costly in other use cases; for example, if the duration of the analysis exceeds the time required to complete client request processing.

In some applications, it is necessary to change the load analysis algorithm dynamically to adapt to new application workloads. Moreover, bringing the system down to reconfigure the load balancing strategy is 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. Application developers often_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 —> the Component Configurator pattern: The Component Configurator design pattern[9] lets applications link and unlink components into and out of an application at runtime. 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 is 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 could 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 Figure 6 depicts.

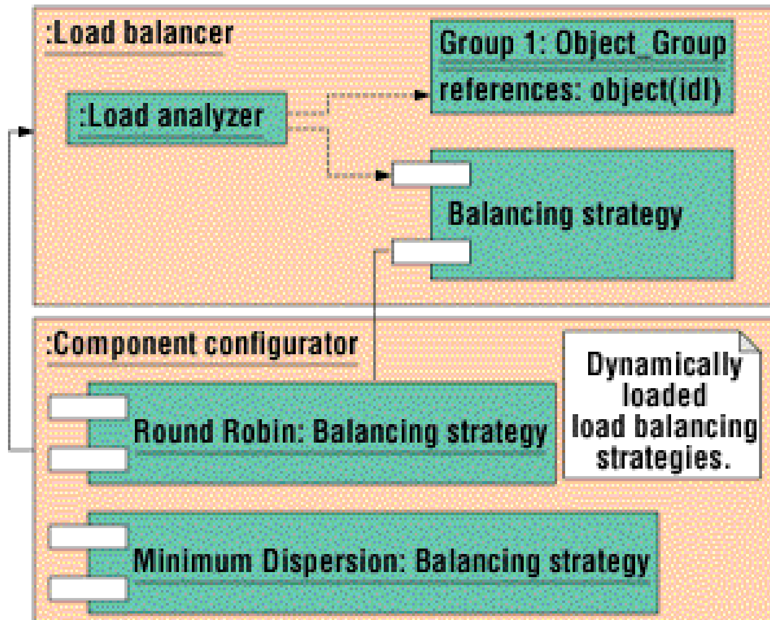



**Figure 6. Applying the Component Configurator pattern to TAO's load balancing service.**

TAO's load balancing service can be configured dynamically to support the following strategies:

> Round-robin: This nonadaptive strategy is straightforward and does not consider load. Instead, it simply causes a request to be forwarded to the next replica in the object group being load balanced.[5]
>
> Minimum dispersion: This adaptive strategy is more sophisticated than the round-robin algorithm. The goal of this strategy is to ensure load differences fall within a certain tolerance. It attempts to minimize the difference between the load on each replica and the average of those loads.

## Challenge 4: Coping with adaptive load balancing hazards

Context: A customized adaptive load balancing strategy is under development. This load balancing strategy will 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 could 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 could begin again, causing system instability.

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

Solution —> Dampening load sampling rates and request redirection: The minimum dispersion load balancing strategy can alleviate the thundering herd phenomenon and balancing paroxysms, because 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 earlier. 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.

## Challenge 5: Identifying objects uniquely

Context: A load balancing service that manages multiple objects is responsible for collecting and analyzing information, such as state, health, and environmental conditions, throughout the lifetime of each object it manages. It obtains this information from the load monitor. 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. We can easily achieve this using 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. (The CORBA Security Service[14] can authenticate client requests, but this is a much more expensive mechanism than required for many applications.) Moreover, as 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 efficiently dispatches processing tasks that result from responses to asynchronous operations a client invokes.[9] 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 Figure 7 illustrates.

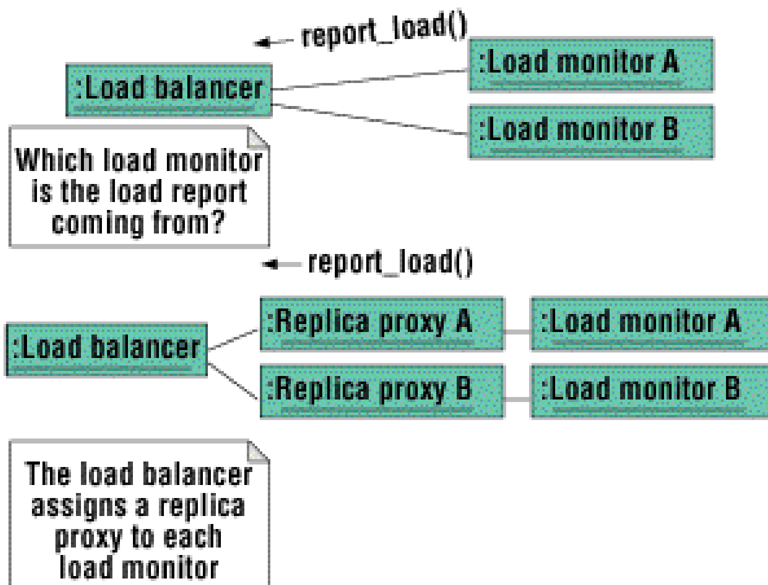



**Figure 7. Identifying the source of a message uniquely.**

The load balancing service creates a unique replica proxy for each monitor. The replica proxy implementation caches the identity of the replica ACT. The load monitor will later push loads to the load balancer through that replica proxy. This design lets the replica proxy determine the identity of the remote replica efficiently whenever new load information is received from the load monitor at the same location as that replica.

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 lets the load balancing service process the requests from each replica efficiently and unambiguously.

As each load is reported to the ReplicaProxy, the load analyzer is notified by the ReplicaProxy that a new load is available for analysis. Because 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.

## Challenge 6: Integrating all the load balancing components effectively

Context: As we illustrated, 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 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 because a given component may be unnecessarily exposed to certain functionality.

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 tie together all the components. It coordinates all interactions between other components—it is a mediator. Thus, clients can remain unaware of the interactions the load balancer mediates, which helps satisfy application transparency requirements.

Applying the solution in TAO: As Figure 1 shows, 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 the load balancer creates to represent 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 the load balancer chooses. The load balancer also rebinds the client to another replica 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 lets the load balancer receive load reports from each registered load monitor. These load reports are then delegated to the load analyzer for analysis. How these loads were obtained is hidden from the load analyzer.

Exploiting the rich set of primitives available in CORBA still requires specialized skills, and requires us to use poorly understood features such as location forwarding. Further research on effective architectures, strategies, and patterns to implement CORBA load balancing services is necessary to advance the state of the art. Our future work will focus on improving our TAO-based CORBA load balancing service described in this article.

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

Acknowledgments

Automated Trading Desk, BBN, Cisco, DARPA contract 9701516, Siemens MED, and the NSF (grant NSF0086094) provided part of the funding for this work.

References

1. Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.4 ed., OMG, Needham, Mass. Oct. 2000.

2. Object Management Group, Persistent State Service 2.0 Specification, OMG document orbos/99-07-07 ed., Needham, Mass., July 1999.

3. D.C. Schmidt et al., "Applying QoS-enabled Distributed Object Computing Middleware to Next-Generation Distributed Applications," IEEE Comm. Magazine, vol. 20, , Oct. 2000.

4. L.R. Welch et al., "DeSiDeRaTa: QoS Management Technology for Dynamic, Scalable, Dependable Real-Time Systems," 15th Symposium on Distributed Computer Control Systems (DCCS98), IFAC, 1998.

5. L. Moser, P. Melliar-Smith, and P. Narasimhan, "A Fault Tolerance Framework for CORBA," Int'l Symp. Fault Tolerant Computing, June 1999, pp. 150–157.

6. Object Management Group, Fault Tolerant CORBA Specification, OMG document orbos/99-12-08 ed., OMG, Needham, Mass., Dec. 1999.

7. D.C. Schmidt, D.L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," Computer Comm., vol. 21, no. 4., Apr. 1998, pp. 294–324.

8. M. Henning and S. Vinoski, Advanced CORBA Programming With C++, Addison-Wesley Longman, Reading, Mass., 1999.

9. D.C. Schmidt et al., Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2, John Wiley & Sons, New York, N.Y., 2000.

10. O. Othman, C. O'Ryan, and D.C. Schmidt, "Strategies for CORBA Middleware-Based Load Balancing," IEEE Distributed Systems Online, vol. 2, no. 3, Mar. 2001, http://dsonline.computer.org/0103/features/oth0103_1.htm.

11. N. Wang, K. Parameswaran, and D.C. Schmidt, "The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware," Proc. Sixth Conf. Object Oriented Technologies and Systems, USENIX, Jan./Feb. 2000.

12. I. Pyarali et al., "Using Principle Patterns to Optimize Real-time ORBs," IEEE Concurrency, vol. 8, no. 1., Jan.–Mar., 2000.

13. E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Mass., 1995.

14. Object Management Group, Security Service 1.8 Specification, OMG Document security/00-03 ed., OMG, Needham, Mass., Nov. 2000.

Ossama Othman is a research assistant at the Distributed Object Computing Laboratory in the Department of Electrical and Computer Engineering, University of California at Irvine. He is currently pursuing his PhD studies and research in the field of secure, scalable and high availability CORBA-based middleware. As part of this work, he is one of the core development team members for the open source CORBA ORB, TAO. Contact him at The Department of Electrical and Computer Engineering, 355 Engineering Tower, The University of California at Irvine, Irvine, CA 92697-2625, ossama@uci.edu.

Carlos O'Ryan is a research assistant at the Distributed Object Computing Laboratory and a graduate student in the Department of Computer Engineering, University of California at Irvine. He participates in the development of TAO, and open source, real-time, high-performance, CORBA-compliant ORB. He obtained a BS in mathematics from the Pontificia Universidad Catolica de Chiled and an MS. in computer science from Washington University. Contact him at The Department of Electrical and Computer Engineering, 355 Engineering Tower, The University of California at Irvine, Irvine, CA 92697-2625, coryan@uci.edu.

Douglas C. Schmidt is an associate professor in the Department of Electrical and Computer Engineering at the University of California at Irvine. He is currently serving as a program manager at the DARPA Information Technology Office, where he is leading the

national effort on distributed object computing middleware research. His research focuses on patterns, optimization principles, and empirical analyses of object-oriented techniques that facilitate the development of high-performance, real-time distributed object computing middleware on parallel processing platforms running over high-speed ATM networks and embedded system interconnects. Contact him at The Department of Electrical and Computer Engineering, 616E Engineering Tower, The University of California at Irvine, Irvine, CA 92697-2625, schmidt@uci.edu.