Using Aspect Oriented Programming to build a portable load balancing service

Erik Putrycz and Guy Bernard
Institut National des Télécommunications
9, rue Charles Fourier
91011 EVRY Cedex, France
{*erik.putrycz, guy.bernard*}*@int-evry.fr*

Abstract

Scaling applications to large networks and an increasing number of users has been since years a technical challenge. Today, technologies are well known to scale applications to local networks but scaling to large networks with high latency is still a challenge.

DLBS (Dynamic Load Balancing Service) brings new solutions regarding large scale load balancing for middleware based applications. DLBS offers a multi-criteria and easily customizable load balancing service. It consists of a scalable monitoring infrastructure, a connection manager (integrated into the middleware) and customizable load balancing strategies.

Implementation of a low level service requires in order to stay efficient to avoid the necessity for high overhead. DLBS aims to be a generic load balancing service and an easy and efficient portability with a CORBA Object Request Broker has been possible thanks to Aspect Oriented Programming.

1. Introduction

Middleware technologies for supporting distributed applications have been the focus of a number of research works, prototypes and product development in recent years, because of their capacities to transparently handle distributed interactions between the various parts of distributed applications and to accommodate legacy applications. As the scope of distributed applications grows (both in terms of geographical spread and number of involved computers), a need for replicating the servers, and for balancing the load induced by the client processes between these many servers, arises.

Load balancing has been first introduced in operating system with process migration mechanism and memory ushering [3]. Process migration and remote execution are ways to delegate to other nodes a process execution in order to reduce its execution time. Those processes are usually schedulable jobs: they are handled by a scheduler and might execute locally or on a remote workstation either immediately or after a delay. Process migration can be driven by a node workload [9, 7], jobs memory [5] and CPU [6] use. As the processes are migrated by the operating system transparently to the applications, there are some limitations on the processes resource use: resources used on a node are not always available on the new node after a migration. MOSIX [3] solves this problem by using the user's "home" node for interactions and resource use. But benefits of load balancing can be lost if the resource use is too high: the network usage becomes a hotspot. Furthermore, this approach assumes that the network latency is very low and consequently, this is not usable on large networks.

Load balancing for large scale networks has been considered mainly in the area of web access, and is usually exercised in the network infrastructure itself. The main characteristic of load balancing in large scale areas is the need to take into account many different kinds of parameters for driving decisions. Latency has a strong impact on the response time of end-user applications but the load on the server may cause huge delays for requests executions. Current solutions integrate load balancing by redirecting messages either by network address translation or by DNS lookups (like Akamai's FreeFlow [1]). But those solutions don't allow the client to react dynamically (during its execution) to the servers load.

The context of our work is client-server applications based on CORBA middleware: they rely on an object request broker for distributed communications. We suppose that clients and servers are separated by a large network (characterized by a high latency and a low bandwidth [2]). As the size of such applications is growing (in terms of distribution, software complexity and resource use) load balancing is necessary to help those applications to scale and keep a good quality of service. For those applications, usual solutions are not suitable for improving their scalability as

Proceedings of the 22 nd International Conference on Distributed Computing Systems Workshops (ICDCSW'02)
0-7695-1588-6/02 $17.00 © 2002 IEEE

show current industry needs for load balancing [11]. To enhance the user experience and make the most of servers load, we introduced load balancing on the client side and with the naming service.

A new challenge arises by developing a load balancing service: how to integrate it into a middleware and still retain portability and efficiency ? Client side load balancing relies mostly on connection management. Connection management is usually located in a very low level in the middleware itself and uses proprietary APIs. In the early versions of our service, we integrated load balancing by doing source code modification in the middleware. This solution was not very satisfactory, not only because of the inherent complexity of a mix of load balancing code and middleware code, but because we were locked to a specific version of the middleware. To solve these issues, we chose to integrate load balancing in the middleware by viewing it as an aspect of the middleware. This means that the link between the middleware implementation and the load balancing service is done using an aspect oriented language. This allows to have a very efficient integration and keep a good separation between our load balancing service and the middleware.

In this paper, we present our load balancing service called DLBS. Our load balancing architecture consists of a monitoring infrastructure for capturing and distributing the load of the server hosts (Section 2). The host load is then used by configurable strategies on different possible reconfiguration points (Section 3). In Section 4, we explain how we used AOP (aspect oriented programming) to make our implementation portable across different off-the-shelf middlewares without compromising efficiency.

2. Scaling monitoring to large networks

The potential of load balancing relies on its capability to dispatch a request to one host chosen between n replicas. To make a wise choice, it is necessary to compare all the hosts in order to find out which one can execute the request the fastest. Without making any hypotheses on external resources use (e.g. databases), we can say that a middleware application requires at least network access, CPU cycles and memory. Network messages for middleware applications are usually short and only the network latency is involved. Therefore, the reactivity of a client application depends mainly on the network latency but if the server is loaded, the request execution time can easily become the hotspot.

DLBS takes into account the computational capabilities (CPU usage and number of threads running), memory (virtual memory use) and network latency (on the client side). Once collected, the monitoring data need to be analyzed and distributed. The usual way to collect information for monitoring is to get data from hosts at fixed intervals (*polling*). It is based on the client-server paradigm, and is not suitable for reducing intrusiveness: the polling rate fixes the precision of the data and therefore the precision is linearly proportional to the number of network messages generated. To scale monitoring to large networks, our monitoring infrastructure relies on two levels of information: a local *Monitoring Registry* and distributed *LoadValues* data (Figure 1). These two levels of information are managed by our *DLBS Agent* (a process running on each server).

The first level of monitoring information (Monitoring Registry data) is used for local storage of low level information. At the second level entities called *Probes* analyze Monitoring Registry data and update the second level data called *LoadValues*. An update of the second level *LoadValues* data propagates the change to all nodes registered in our Monitoring Agent listening to this host load. A change in *LoadValue* data is broadcast using simple UDP messages: they have the lowest possible network cost and their loss is acceptable. At a first sight, the amount of traffic generated may look costly. But by diminishing the reactivity, those messages will only be sent on important changes and because they contain only monitoring data, they have a small size (our monitoring message size is only 190 bytes).

3. Dynamic Load Balancing Architecture

Now that significant data for system performance are determined and we have an infrastructure to distribute them, the next challenge is to use those data to drive load balancing. Load balancing can only be done dynamically under some constrains (Section 3.1). The dynamic characteristics are achieved with the reconfiguration of the middleware itself. We offer reconfiguration in many different locations of the middleware to suit different levels of dynamicity (Section 3.3). All load balancing decisions are driven with strategy modules (Section 3.2). The setup of load balancing itself is eased through our replicas and monitoring agents management (Section 3.4).

3.1. Constraints on stateful servers

On most middleware, a server state is bound to a client session and so enabling load balancing during a session implies to have a coherent state on each replica. This can be achieved through replication mechanisms but they are usually costly and complex to manage. This is why enabling load balancing during a session has to be taken into account when designing the application: all the state bound to a session should be stored on the client side. Therefore, we provide two different kind of reconfiguration: static (done on a session basis) and dynamic (done on the request basis).

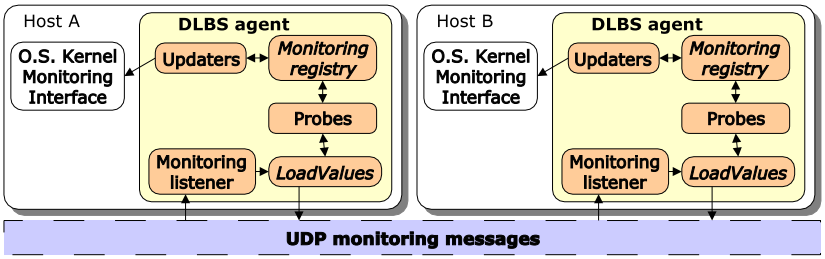

Figure 1. Distributed monitoring architecture

3.2. Load Balancing Strategies

Most load balancing services don't separate the load balancing framework (monitoring, connection management) of the load balancing decisions. For single criteria load balancing, the need to customize the strategy may not be necessary. With many monitoring criteria, the way each criteria is taken into account in the decisional process has to be customizable from an application to another (an application which requires lot of computation has to rely mostly on CPU load instead of many other applications whose reactivity depends mainly on the latency). To ease the construction of the load balancing strategies, DLBS drives the decision using *strategy modules*. Each strategy is build around a `processStrategy` method. We offer APIs for accessing LoadValues and setting preferences for host. A `processStrategy` method has then only to give the preferred hosts for a specific criteria. With our model, a CPU based strategy can be written in few lines. This provides an easy way to adapt load balancing to any application: a configuration files contains all the strategies which are taken into account and a weight associated with each strategy.

3.3. Reconfiguration Points

Using monitoring data for driving the choice between a pool of replicas implies to reconfigure the application by modifying client-server bindings. Those bindings can be modified inside locations that we call *Reconfiguration Points*. Their locations depend heavily on the brokering capabilities and for a CORBA based middleware, we provide two different locations: naming service and client.

Reconfiguring on the naming service is the most easiest way to integrate load balancing in an existing application. When the client issues a `resolve(...)` call to the naming server (to get a server reference by name), we send the reference of the hosts sorted by the load balancing strategy. This is a static way of reconfiguration (session based) but there are no restrictions on sessions of the server objects. But a static way is suitable only for short-lived clients because they don't need to adapt to the servers load during their session.

The latest reconfiguration way in our infrastructure is the client-side reconfiguration. Client-side reconfiguration is fully transparent and doesn't require any modifications on existing client code. It is managed by a lightweight version of the server side framework (there are only the Monitoring listener, StrategyManager and ReplicaManager) which are created and integrated inside the middleware. In this scheme, the client is running a Monitoring data listener and takes decisions on its own (the load balancing strategies are located on the client-side). This provides the most reactive way as there is no intermediate host for driving the policies. Thus, client-side reconfiguration is more suitable for long lived clients and high latency networks.

Figure 2 shows a scenario of client reconfiguration in the case of a simple CORBA server responding to a `hello()` request. When a `hello()` request is issued by the developer client application, the stub transforms it into a generic `invoke(...)` call and delegates it to the `ClientDelegate`. Then, before sending the request to the network, it goes through our `ConnectionManager` class to check for a reconfiguration binding of the destination. The UDP listener for `LoadValues` data is running on the client-side and runs the load balancing strategy when it receives an update of the `CPU LoadValue`. After an update, the strategy manager creates a reconfiguration binding of the original server A (reference given by the naming service) to the less loaded host (in this case server B) given the strategy processing. All other `invoke(...)` calls will then be issued to server B.

3.4. Replicas and monitoring agents management

A important concern of load balancing is to ease its use enough to be fully handled by a system administrator at the deployment of an application. Fundamentally this means that the setup of a replica has to be fully non-functional. Most Object Request Brokers only offer interfaces to create groups of replicas and can be used by a trained developer only. The creation of replicas group and the setup of load balancing have to be manageable by tools (as Microsoft Application Center [10] and BEA's Weblogic [4]). For instance, Microsoft Application Center allows to gen-

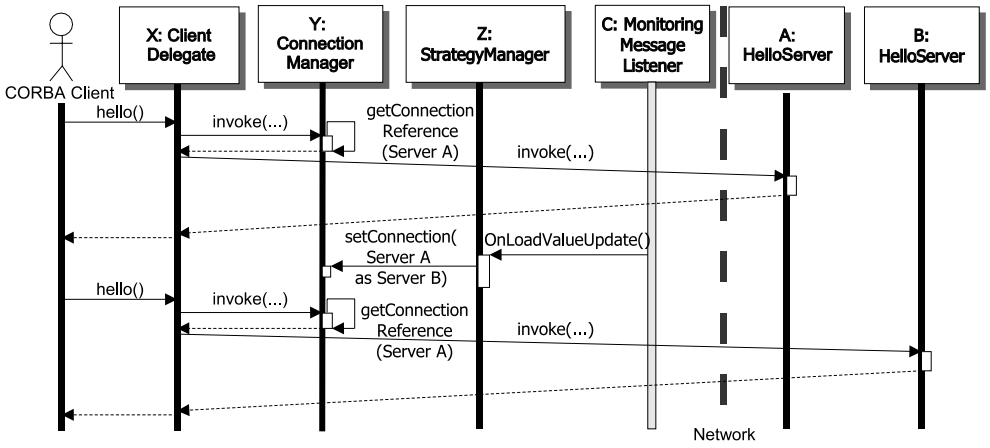

Figure 2. UML Client Reconfiguration Sequence Diagram

erate deployment profiles to install and execute the server applications on many hosts. Then, the load balancing service can be setup thought the management console since the group creation is handled transparently.

We choose a simple scheme to transparently create all the group of replicas and enable load balancing: we modified a CORBA Naming Server in order to allow multiple registrations under the same name. The first registration for a name is the master of a replicas group. The master is in charge of managing the group. The group is created itself at the second registration. Then, all next registrations under the same name add hosts to the group list. In case of client-side reconfiguration, every replica will need to know the client list. So joining a group of replicas propagates the client and replicas list to all hosts. These functions are located in a CORBA object called *ReplicaManager* running alongside each server.

The UML sequence diagram of Figure 3 shows a sample execution case when a replica called server B is registering itself with the same name as an existing server (server A). The CosNaming Listener (part of the DLBS framework) running on the same host than the naming service catches every CORBA `rebind(...)` call (used to register a server in the Naming Service). When there is already a reference registered under the same name a replica group is created. The new replica registration is then propagated to all the hosts of the infrastructure.

In case of client-side reconfiguration, the client needs to have information about the group of replicas. This information is transmitted transparently to the client at its initialization phase using CORBA multi-profile object references.

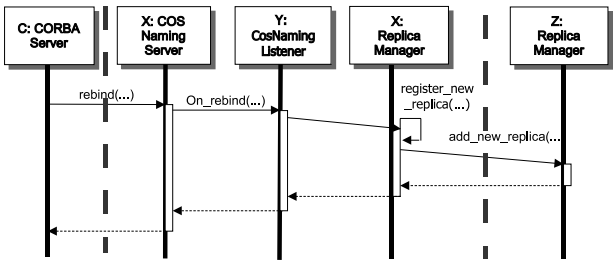

Figure 3. UML sequence diagram of a new replica group creation

4. Enhancing portability with AspectJ

4.1. Integrating the service with an existing middleware

To achieve the lowest possible overhead, we chose to integrate it on the lowest possible level. In the first version of our load balancing service, we integrated all the replicas management and load balancing decisions into the broker by modifying the broker source code. Those modifications are complex to do and to maintain: on client and server side we have to play with real destinations of requests and this is usually hidden in the CORBA IOR (Interoperable Object Reference), its modification requires to use lot of low level and proprietary APIs (IOR are standardized into CORBA specifications but not the APIs for their access and manipulation). So it was not possible to cleanly separate the DLBS source code of the proprietary code making impossible to change the broker version. It made very hard to integrate the service from an implementation of CORBA to another because of having to make low level modifications.

The choice of a low level integration makes not possible

to achieve a full portability without any modification (IOR access and manipulation require to use proprietary APIs). To go towards portability, we localized the most possible the source code accessing the proprietary APIs. We will detail how we integrated load balancing with ORBacus (a well known implementation from IONA of the CORBA 2.4 specifications).

4.2. Aspect Oriented Programming with AspectJ

Aspect-oriented Programming is a technique for improving the separation of concerns in software. It provides language mechanisms that explicitly capture crosscutting structure [8]. Since our implementation is done with Java, we chose to use AspectJ for creating aspects.

AspectJ is an aspect-oriented extension to the Java programming language. It supports dynamic and static crosscutting. The key concepts in AspectJ are *Joint Points*, they are well-defined points in the execution of a program; *pointcuts* are a means of referring to collections of join points and certain values at those join points; *advice* are method-like constructs used to define additional behavior at those join points; and *aspects* are units of modular crosscutting implementation, composed of pointcuts, advice and ordinary Java member declarations.

We used AspectJ by creating an AspectJ *aspect* for two kinds of integration:

- Client side load balancing (*aspect CSLoadBalancing*) ;
- Naming Service load balancing (*aspect NSLoadBalancing*).

By this way we localized all the interactions of the generic load balancing service code with the ORBacus specific code in two AspectJ files and few plain Java classes. Those files consists of 500 lines approximately which ease a lot to maintain and debug our service compared to direct integration (ORBacus source code is around 60000 lines). Another AspectJ feature which eased a lot the integration is the possibility to have a state associated with an *aspect*. This allows to access the references of the main entities of our framework in each advice (compared to the complexity of passing reference everywhere).

AspectJ allows a very effective integration: the overhead induced by AspectJ is very low. The overhead of the load balancing service is itself so low (around 1ms) that it's nearly unnoticeable and makes the AspectJ overhead very hard to measure.

4.3. Client Side Load Balancing

Integrating load balancing on the client side consists of:

- creating an instance of our service to receive monitoring data ;
- detecting load balanced servers ;
- catching requests and if necessary, redirecting them to the host chosen by the Load Balancing strategies.

The instance of our load balancing and monitoring framework is created on the launch of the client. We used an AspectJ *after* advice to catch the creation of the Object Request Broker and instanciate our framework after its creation. The detection of load balanced servers is done on connection creation and if the reference contains multiple profile, we suppose that each reference is a replica. This has been implemented using a *around* advice on the connection creation.

The most challenging issue is to redirect requests from a host to another. This redirection requires to involve the `ReplicaManager`, the Load Balancing `StrategyManager` (which relies on the monitoring service) as the connection manager (which manages associations between the proprietary connection object and our `Location` object used in strategies). On ORBacus, connections are managed with `ClientProfilePair` objects. Connections (`ClientProfilePair` objects) are cached and we inserted the load balancing strategies on the access to the cache. The access to the cache is called `getClientProfilePair` and we change the effective destination using an *around* advice on this method to return the less loaded host (Figure 4) given by the strategies processing.

4.4. Providing load balancing to the naming service

The integration with the CORBA naming service is done by maintaining a copy of the naming service state (name and objects associations). With AspectJ *around* advises, we catch every call to `bind`, `unbind` and `resolve` and control their execution. By this way, we can allow many `bind` calls with the same name. A second call to `bind` bypasses the standard mechanism and create a replica group. We have an *around* advice on the bind method. Firstly, we get the client IP then, if this `bind` call is not the first one with the same name (given by the return value of `_cnl.onCosNaming_bind`) we bypass the standard execution (which is done with `proceed`).

5. Conclusions

As distributed applications use is growing, a key challenge is to help them to scale to many users and to large networks. It is possible to enhance a user's experience by driving load balancing with monitoring data to make the

```
package dlba.orb.orbacus;
aspect CSLoadBalancing { ...
com.ooc.OB.Client around(com.ooc.OCI.ProfileInfoHolder profileInfo,
  com.ooc.OB.DowncallStub ds): target(ds) && args(profileInfo) && call(com.ooc.OB.Client
  com.ooc.OB.DowncallStub.getClientProfilePair(com.ooc.OCI.ProfileInfoHolder)) {
    com.ooc.OB.Client default_client = proceed(profileInfo,ds);
    String op_name = (String) OpNameByDCS.get(ds);
    if (op_name != null) OpNameByDCS.remove(ds);
    org.omg.IOP.IOR currentIOR = (org.omg.IOP.IOR)IORbyDCS.get(ds);
    //_ccm is the JOBClientConnectionManager instance
    // onInvoke returns the host chosen by all the load balancing strategies
    com.ooc.OB.Client new_client =
      _ccm.onInvoke(profileInfo, currentIOR, default_client, op_name);
    return new_client; }}
```

Figure 4. Client Side AspectJ code

most of non loaded hosts. Current middleware load balancing solutions are based on simple load balancing strategies and most work on monitoring-based load balancing are not adaptable to modern applications. Our innovation in the technology available for large scale load balancing resides in the following characteristics. First, our mechanism is able to dynamically reconfigure the client given monitoring data (in the middleware itself, that is in a transparent way to applications). This approach has several benefits: (i) the most "appropriate" server instance can be defined according to our customizable and configurable load balancing strategy model, and not according to a blind mechanism, or strategies forged into the service itself ; (ii) there is no need for pre-processing every client request on some centralized place, which would always be at the wrong place. Second, geographical scalability is achieved through an asynchronous (event) model for propagating significant load changes in the state of the servers to the clients (cache mechanism), whereas current solutions involve a supplementary indirection through dispatchers for every client request.

Achieving portability has been possible thanks to AOP and AspectJ. Portability of a load balancing service is a challenge because very often portability implies to add many software layers for adaptation which are costly on execution (e.g. CORBA Portable Interceptors). Adaptation in the source code through making load balancing an *aspect* of the middleware not only allowed us to keep the service efficient, but it enhances and eases the software maintenance and debugging by localizing all the code specific to a port into few files.

References

[1] Akamai. How freeflow works. http://www.akamai.com/.

[2] G. Ballintijn, M. Van Steen, and A. S. Tanenbaum. Characterizing internet performance to support wide-area application development. *Operating Systems Review*, Oct. 2000.

[3] A. Barak, O. La'adan, and A. Shiloh. Scalable Cluster Computing with MOSIX for Linux. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, 1998.

[4] B. Corporation. *BEA WebLogic Server Administration Guide*. BEA Systems.

[5] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, 1999.

[6] M. Harchol-Balter and A. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[7] C. Hui and S. Chanson. Improved strategies for dynamic load balancing. *IEEE Concurrency*, 99:58–67, July-September 1999.

[8] G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Grisworld. An overview of aspectj. In *ECOOP '01 - Object-Oriented Programming*, Budapest, Hungary, June 2001. Springer-Verlag.

[9] R. Lavi and A. Barak. The home model and competitive algorithms for load balancing in a computing cluser. In IEEE, editor, *Proceedings of The 21st IEEE International Conference on Distributed Computing Systems*, Mesa, Arizona, Apr. 2001.

[10] Microsoft Corporation. Microsoft application center 2000 component load balancing technology overview. http://www.microsoft.com/technet/acs/clbovrvw.asp, Sept. 2000.

[11] Object Management Group. Load balancing and monitoring for corba-based applications request for proposals. orbos/2001-04-27, Apr. 2001.