Design and Implementation of a Flexible Load Balancing Service for CORBA-based Applications

Markus Aleksy, Axel Korthaus, Martin Schader
University of Mannheim, Germany
{aleksy|korthaus|mscha}@wifo3.uni-mannheim.de

Abstract

In this paper we present a flexible load balancing service for CORBA-based applications. It offers several static and dynamic strategies for load balancing. The extensive adaptability and flexibility enable usage of the service in a wide variety of problem domains. Analogies to CORBA's Trading Service have the benefit that developers that have worked with this service are soon familiar with our architecture.

1. Introduction

CORBA-based applications have gained increasing popularity since version 2.0 of the CORBA standard was published. CORBA 2.0 specified the Internet Inter-ORB Protocol which enabled interoperation of clients and servers that are implemented with ORBs of different vendors.

In addition to the core architecture [4], the CORBA standard defines several services. These services augment and complement the basic performance of an ORB by specifying often needed functionality like, e.g., a Persistence Service, a Name Service, an Event Service, etc. Standardizing the IDL-interfaces of the services increases portability of applications relying on them. Although many aspects of application development are supported, a Load Balancing Service is not defined in the latest CORBA specification. To remedy this deficiency, we implemented the Load Balancing Service discussed in the following.

2. Design Goals

The design goal during development of our service was not centered on obtaining the highest possible performance. Instead, we concentrated on the following architectural aspects:

- Portability. Only minimal vendor- or platform-specific features should be used.
- Flexibility. The architecture should be suitable for a large variety of applications. This implies that each component supports different strategies and that users can select the strategy they prefer.
- Adaptability. It should be possible to configure the functionality of the service easily to the characteristics of the respective application.

3. The Load-Balancing Service (LBS)

Like all other CORBA services, the LBS was conceived as an independent central component [2]. This decision involves several advantages but also some disadvantages. On the positive side we can mention:

- information gathered on server load is centrally accessible and needs not be distributed or replicated in the network,
- the LBS is equipped with information on the complete application and, thus, can prevent counter-productive decisions,
- with a centrally controlled assignment of requests to servers a rather uneven distribution of tasks can usually be prevented.

Some of the disadvantages should also be noted:

- recording of load information additionally burdens the LBS,
- as a central component, the LBS is not or not fully scalable; therefore, depending on the number of concurrently incoming requests, the LBS might be overloaded and through the resulting delays become a bottleneck,
- as a central component, the LBS constitutes a single point of failure; its breakdown causes the failure of all connected applications.

The first criticism could be met by delegating the recording of load information to the individual servers. Here, the term “servers” denotes not only the different servers but also other services whose load needs to be recorded. Assigning load information retrieval to the servers somewhat slows down their performance but it considerably relieves the LBS since it is a central component.

The next two disadvantages can be overcome if—at runtime of the applications—we start several LBS instances and register them with the Name Service [5] or Trading Service [6]. This approach, however, increases expenses on the side of the clients who now have to obtain and examine different names before submitting a request.

At present, the LBS consists of two main components:

- an information gathering component (`ServantMonitor`) and
- a load balancing component (`Balancer`).

The `ServantMonitor` is responsible for gathering statistical data. It provides a standard query interface (`MonitorDataPool`) which the `Balancer` and the clients can access. Determination of the load parameters can be based on the total load of a server or individually on each CORBA object (servant). In that context the interface `MonitorManager` plays an important role. It is part of the `ServantMonitor`, supplies it with the configuration parameters used, and informs it on the start of the monitoring process.

The `Balancer` performs two different tasks. First, it dispatches incoming requests to the servers. On the other hand, it works as a *Locator* that sends a complete list of eligible servers to the caller without making any scheduling decision on its own. In the *Locator* role the `Balancer` does not analyze load data and is therefore noticeably less loaded than in the first case. The caller itself now has to decide to which server the request should be passed. A different server might be used for each request. This, however, will only make sense if the requests do not change the server’s state since, otherwise, inconsistencies might occur.

3.1. Strategies supported by the LBS

The `Balancer` supports a variety of different load balancing algorithms. These can be classified into two basic types:

- static algorithms and
- dynamic algorithms.

In contrast to dynamic algorithms, static algorithms do not rely on information on system state. Therefore, they do not achieve the best results but their resource consumption is limited.

The following static algorithms that produce little overhead are implemented in the LBS:

- `random`. Here, the servant is randomly selected.
- `round_robin`. One servant after another is selected for processing client requests.
- `longest_unused`. The servant that was not used longest is selected.

Examples of dynamic algorithms supported by the LBS are:

- `longest_idle`. The servant that was idle longest is selected.
- `highest_idle_percentage`. The servant that was idle most frequently is selected.
- `fewest_request_per_second`. The servant that processed the fewest number of requests per second is selected.
- `shortest_request_avg`. The servant with the smallest average processing time is selected.
- `shortest_request_min`. The servant with the smallest minimal processing time is selected.
- `shortest_request_max`. The servant with the smallest maximal processing time is selected.
- `fewest_clients`. The servant with the smallest number of completed results is selected.
- `best_mix_index`. The servant with the smallest ratio of average processing time by processing percentage is selected.

The architecture of the LBS shows several conceptual and structural analogies to the Trading Service. The reason is, that both services accomplish similar tasks in gathering and distributing information. The reuse of structures known from the Trading Service also has the benefit that developers that have worked with this service are soon familiar with the LBS.

3.2. Measuring Performance using CORBA

In principle, portable performance results concerning servant load can be obtained with two techniques:

- with an Interceptor or
- by using a Servant Locator.

Interceptors provide a callback mechanism that provides interested parties with the means to observe and intercept the communication process between client and server. They can be used, e.g., to authenticate a client, to keep a record of requests, or for debugging purposes. The location where an Interceptor is interposed in the invocation path between client and server is called interception point. By inserting interception points we can set up measuring points where data needed for performance considerations may be obtained. A drawback of this technique is that interceptors were first specified in CORBA 2.2 and are, consequently, not supported by all products available now. At the moment, existing ORBs can be classified into three categories:

- ORBs without Interceptor implementation,
- ORBs with non-standard Interceptor support, and
- ORBs with standard conforming Interceptor support.

Due to reasons of portability, we decided not to rely on this technique. Further experiences with CORBA Interceptors are reported in [7].

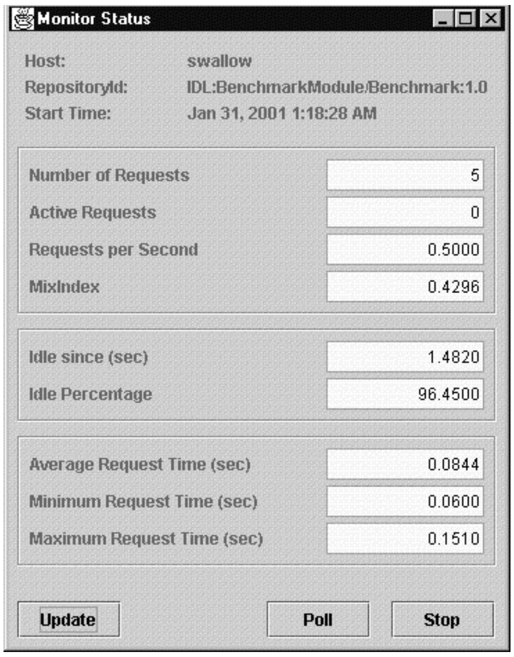
The second possibility was to rely on a Servant Locator. With the specification of the Portable Object Adaptors (POAs) the OMG has introduced several new fine-grained concepts and notions. A CORBA object, now, is a virtual entity with an identity and an interface. On the other hand, a servant is a programming language entity that implements a CORBA object and exists within the context of a server process. The assignment of a CORBA object to a servant is called incarnation. Incarnation is essential for the activation of an object which enables it to process requests. Activation can be accomplished in different ways:

- explicitly by calling `activate_object()`,
- implicitly by creating the POA with the `IMPLICIT_ACTIVATION` policy, or
- by associating a Servant Manager with the POA.

The last approach is the most interesting for the purpose of performance measurement. It enables an application to register a callback object (the Servant Manager) with the POA. Each time the POA receives a request targeted at an inactive object the Servant Manager is called. Two types of Servant Managers are defined: Servant Activator and Servant Locator. The Servant Activator is used for the persistent activation of an object. Contrarily, a Servant Locator returns an object that will be used for a single request. This implies that the Locator will be involved in each operation call. The Servant Locator interface defines two operations `preinvoke()` and `postinvoke()` which are always called in pairs. The operation `preinvoke()` is invoked by the POA whenever it receives a request for an inactive object. `preinvoke()` returns a servant which will be used to process the request. Additionally, through an out parameter, a Cookie is returned which - after the invocation of the request on the servant is complete - can be used to identify the corresponding `postinvoke()` call. The `postinvoke()` operation is invoked when the servant completes the request. In this way - measuring the time difference between the `preinvoke()` and `postinvoke()` calls - we can determine how long the servant was occupied with processing the request. No system-specific features are accessed here.

3.3. The Time Measurement Component

As described above, the task of recording request processing times was assigned to the servers in order to relieve the LBS. To save developers time measurement-related work, the LBS contains a dedicated component. The class `ServantMonitor_impl` offers not only the complete functionality needed for time measurement but also a user interface (cf. Figure 1) which displays the actual data. This interface can be employed for purposes of load control and for performance optimization.



Host:	swallow
RepositoryId:	IDL:BenchmarkModule/Benchmark:1.0
Start Time:	Jan 31, 2001 1:18:28 AM
Number of Requests	5
Active Requests	0
Requests per Second	0.5000
MixIndex	0.4296
Idle since (sec)	1.4820
Idle Percentage	96.4500
Average Request Time (sec)	0.0844
Minimum Request Time (sec)	0.0600
Maximum Request Time (sec)	0.1510

Figure 1: The Graphical User Interface of `ServantMonitor_impl` component

3.4. Locating a Servant

Three techniques can be applied by a client to obtain a servant's reference:

- by entering the servant name as a string,
- by conducting a search based on servant characteristics, or
- by specifying the desired servant type.

The first approach is very user-friendly but, typically, clients have no clue on the names that might be used by the servant. A search based on servant characteristics seems rather flexible. However, in the case of an LBS it is often time consuming and causes too much overhead. The last possibility seems to be the most suitable since in most cases a client has to "know" the type of the servant in order to be able to use its services.

3.5. The LBS from the Perspective of a Client

We have already mentioned that a client can request the LBS to provide exactly one object reference. And it can suggest different policies that affect the way this reference is obtained. It can instruct the LBS to use only static algorithms, to evaluate a given quantity of load data, to provide a specified server exclusively for itself, etc. Whether these suggestions are accepted or not is determined through the LBSs configuration. This configuration is entered during LBS startup and usually cannot be modified afterwards. A differing load balancing variant can be achieved by importing different references of the desired server type. Here, the client itself determines the load of the servers, selects the best suited server or servers and sends its request.

4. A Code Example

To register with the LBS a server, first of all, has to obtain a reference to the LBS. The class `ClientUtil` provides the methods needed for that purpose. It attempts to gain access to the LBS via a request to the Name Service [5]. In a second step, the time monitoring component (`ServantMonitor_impl`) must be generated and initialized. Finally, the server reference together with the monitor reference as well as some additional information is exported. The following Java code snippet illustrates this procedure:

```
// ORB orb = ...
// POA poa = ...

MyObject_impl = new MyObject_impl();

Balancer balancer =
  ClientUtil.resolveBalancer(
    orb, "LB1", "");

ServantMonitor_impl monitor =
  new ServantMonitor_impl(
    myObject_impl, balancer, orb, poa);


monitor.set_repositoryId(
  myObjectHelper.id());

org.omg.CORBA.Object servantRef =
  monitor.get_servant_reference();
MonitorDataPool datapool =
  monitor.datapool();
MonitorManager manager =
  monitor.manager();
poa.the_POAManager().activate();

String extMonitorId =
  ClientUtil.export(myObject_impl,
    servantRef, datapool, manager);
```

It can be seen that this solution implies only slight modifications on behalf of the server. The modifications needed on the client side are also insignificant. Thus, existing applications can be easily integrated with the LBS.

5. Related Work

In [1] a different approach for load balancing is proposed which is very interesting from the perspective of the client. There, the Name Service is enhanced with load balancing capabilities. As a result of calling `resolve()` a client receives the IOR of the least loaded server registered with the given name. This, however, implies that different servers can register with the same name which is not standard-compliant since an `AlreadyBound` exception should be raised in that situation.

The Load Monitor developed at the University of Frankfurt [3] is a further example of a CORBA-based tool for load determination. In contrast to our proposition, there, system-specific Unix interfaces like, e.g., `/dev/kmem` and `/dev/kstat` are utilized to gather load data.

6. Critical Review

With the help of the LBS it is possible to close a gap in the realm of the currently specified CORBA services. Developers are now able to provide their applications with load balancing abilities.

When relying on the LBS, developers can influence the selection of the resulting server through different criteria. Static or dynamic algorithms can be applied. Should these criteria be insufficient one or more servers can be requested for exclusive use. It is also possible to request a list of references of all available servers and to select one or more servers from that list. The extensive means for configuration enable users to flexibly apply the service to a large variety of problem domains.

Analogies to the Trading Service will shorten the learning phase for developers that have experience with that CORBA service.

During development of our LBS prototype we refrained from employing vendor-specific extensions to the tested ORBs whenever possible. As an implementation language we used Java. Thus, an easily portable solution was created.

References

[1] Barth, T., Flender, G., Freisleben, B., Thilo, F. (1999): “Load Distribution in a CORBA Environment”, in: Proceedings of the International Symposium on Distributed Objects and Applications, 5-6 September 1999, Edinburgh, Scotland, IEEE, Los Alamitos, California, pp. 158-166

[2] Hoffmann, Michael (2000): “Entwurf und Implementierung eines CORBA-basierten Load-Balancing Service”; Master Thesis, Department of Management Information Systems III, University of Mannheim

[3] Geihs, K., Gebauer, C. (1997): “Load Monitor LM – Ein CORBA-basiertes Werkzeug zur Lastbestimmung in heterogenen verteilten Systemen”, in 9. ITG/GI-Fachtagung MMB’97, “Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen”, TU Bergakademie Freiberg/Sachsen

[4] OMG (2000): “CORBA/IIOP 2.4 Specification”; OMG Technical Document Number 00-10-01, ftp://ftp.omg.org/pub/docs/formal/00-10-01.pdf

[5] OMG (2000): “Naming Service Specification“; OMG Technical Document Number 00-06-19, ftp://ftp.omg.org/pub/docs/formal/00-06-19.pdf

[6] OMG (2000): “Trading Object Service Specification“; OMG Technical Document Number 00-06-27, ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf

[7] Wegdam, M., van Halteren A. (2000): “Experiences with CORBA interceptors“; Workshop on Reflective Middleware (RM2000), 7-8 April 2000, New York, http://www.comp.lancs.ac.uk/computing/rm2000/papers/20-aacentcweg.pdf