Published in 12th International Conference on Parallel and Distributed Computing Systems (PDCS'99), 7580-7585, 1999
which should be used for any reference to this work

The Design of a Trader-based CORBA Load Sharing Service

ELARBI BADIDI[1], RUDOLF K. KELLER[1], PETER G. KROPF[2], VINCENT V. DONGEN[3]

[1]Départ. Info., Université de Montréal,
Québec, Canada
{badidi,keller}@iro.umontreal.ca

[2]Départ. Info., Université Laval,
Québec, Canada
kropf@ift.ulaval.ca

[3]Centre de Recherche Informatique de Montréal,
Québec, Canada
vandonge@crim.ca

Abstract*

In the context of CORBA, standard services for naming, transaction processing, and alike have been defined. However, no standard support is provided to handle load imbalances that may occur in distributed object applications where several object servers are used to provide the same service type. To address this shortcoming, we have designed a CORBA-compliant load sharing service (LSS). Preliminary experiments with LSS show that LSS allows for the uniform load distribution among servers. Furthermore, we report that load sharing via LSS performs better than the approaches based on the random and the round robin server selection methods, in terms of the average response time for client requests.

Key words: *CORBA, load sharing, dynamic server selection, server discovery.*

1. INTRODUCTION

Today's distributed services are deployed in wide area distributed systems, such as the Internet, and serve a large population of users. Many of these services require a high degree of availability regarding both, computer failures and network failures. The deployment of such services is becoming increasingly easy, specifically, with the advent of Distributed Object Computing environments, such as CORBA [11] and DCOM [9]. Object servers in these environments may serve various clients written in different languages. Therefore, distributed systems have to scale in terms of both, the number of services and the number of users. In this context, availability and performance are very significant issues.

CORBA provides basic mechanisms for remote invocation through the object request broker (ORB), as well as a set of services for object management [10]. Nevertheless, neither the ORB nor the existing services provide tools for balancing the load among the objects of a distributed application that requires a high degree of availability. In this paper, we focus on a particular aspect of the problem: given a specific service type, how can we locate servers providing this service type, and how can we distribute service requests in a way that minimizes the average response time for client requests and increases availability of these servers?

As a solution to this problem, we present the design of a CORBA compliant load sharing service (LSS). LSS has been designed with scalability issues in mind. To take advantage of this service, client and server objects should inherit from specific classes of the service. A preliminary series of tests shows that the use of LSS results in a uniform load distribution. The average response time of client requests is considerably smaller with an increasing number of requests, when compared to random and round robin server selection.

The remainder of this paper is organized as follows: in Section 2, we present some background about load sharing and server discovery in CORBA. In Section 3, we expose the design of a CORBA load sharing service by presenting the service structure and two utilization scenarios. In Section 4, we present experimentation results. In Section 5, we review some related work, and finally, in Section 6, we conclude the paper by discussing future work.

2. BACKGROUND

2.1. Load sharing in CORBA

The present work has evolved from our previous work described in [3] where we propose an architecture, called LoDACE, which allows load sharing between object servers in an object-based distributed system. Load sharing is achieved through the initial placement of client requests among available servers that provide the required service. One drawback of the LoDACE architecture is its centralized design. In fact, the load manager collects load information from all registered servers, and is responsible

* This work is being supported by the National Science and Engineering Research Council of Canada (NSERC), under grant number OPG0175039, and by Centre de Recherche Informatique de Montréal (CRIM).

for finding least loaded servers. Thus, it may become a bottleneck with an increasing number of clients and servers in the system. This has led us to consider a distributed approach by shifting from the design of an architecture to the design of a CORBA service for load sharing, which is compliant with the CORBA object service philosophy. The service classes can be inherited by clients and servers to implement load sharing. Clients can inherit certain classes to be able to get their required services from least loaded servers. Servers can inherit other classes to be able to export their services and share load with other servers providing the same service type. No central component is needed for gathering load information and finding least loaded servers, thus providing a scalable service.

2.2. Overview of the OMG Trading Service

The *trading service* is a service that allows clients (or users) to dynamically discover offered services. The object providing this service is called a *trader*. A trader is a mechanism that facilitates the advertisement and the discovery of services. It communicates with servers, with customers, and with other traders [10]. When a server wishes to announce its service, it records its offer in the trader's database. An offer of service contains a type of service, an identifier of the interface of the service where that service is provided, and the values of the service properties. The advertisement of a service is called *exportation*. When a customer requires a service, it sends a request of service to the trader to find a suitable service. A request of service expresses the characteristics required by the customer by specifying the type of the desired service and the constraints on the service properties. This operation is called *importation*. The trader, then, searches its database for service offers that match the customer's request. The list of the service offers found is then returned to the customer, which can choose from this list a server to which its service requests are to be sent.

3. A CORBA LOAD SHARING SERVICE

The LSS service is intended to be generic and not specific to certain object types. Load sharing is meaningful only if applied to object servers providing the same service type. The goal of this service is to provide programmers with the ability to share the load between several object servers providing the same service type. Thus, client service requests will be assigned to lightly loaded servers. The expected results are improved response time for clients, and increased availability of servers.

The set of objects concerned with load sharing is dynamic since objects can go up or come down dynamically. This complicates the task in comparison with systems that perform load sharing between the processors of a distributed system where the configuration of nodes is static. Consequently, load sharing in distributed object environments requires dynamic discovery of object servers providing the same service type.

The LSS service presented here is based on the use of the OMG trading service. The design of LSS distinguishes the following categories of objects: 1) **Client objects** are objects interested in getting a certain service type. Using LSS, they will be able to discover object servers and to send their service requests to least loaded ones. 2) **Application objects (servers)** are objects providing different service types. These objects are application domain dependent. Object servers providing the same service type form a cluster of objects for which load sharing is to be performed. To take part in the load sharing process, object servers have to announce the service types they provide and their load should be monitored. 3) **LSS Internal objects** are objects which allow: (i) client objects to discover least loaded object servers providing their required services, (ii) objects servers to export, update, or withdraw their service offers, and (iii) to monitor object servers load.

3.1. Service structure

Figure 1 illustrates the LSS class diagram, which shows the LSS interfaces and the relationships between them. These interfaces provide different views for the users of the LSS service:

1. The client's view to find suitable object servers and least loaded ones,
2. The object server's view to advertise services and to monitor servers load,
3. The service's view used by LSS internal objects to invoke application object servers.

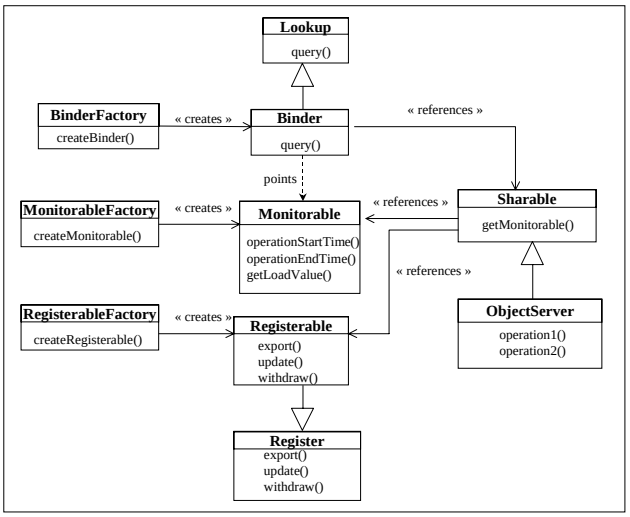

Figure 1 - Class Diagram of the LSS Service

Figure 2 presents these views and the interfaces used by each object of the system.

The client's view

This view allows a client to dynamically discover object servers which provide the required service according to its constraints about the service properties, and which present least load. Once a target object server is discovered, the client can invoke its methods. Object servers are discovered using the Binder interface. Binder objects are created by invoking the *createBinder()* method on a BinderFactory object. The Binder interface defines an operation called *query()*, which is inherited from the trading service Lookup interface. The effect of a call to the Binder *query()* method is to forward the request to the Lookup interface and then to determine least loaded servers among the list of servers returned as the result of the Lookup *query()* operation.

The object server's view

This view allows to monitor the load of object servers and to advertise their services. Load monitoring of an object server requires that it creates a Monitorable object by invoking the *createMonitorable()* operation on a MonitorableFactory object. A Monitorable object provides three methods: (1) the operation *operationStartTime()* is used by a server to report the arrival of an invocation on one of its objects, (2) the operation *operationEndTime()* is used by a server to report the end of processing a request, and (3) the operation *getLoadValue()* that is used by a Binder object to get the load value of the object server associated to a Monitorable object. The Monitorable object maintains a history about service requests made to the associated object server.

To advertise its service offer, an object server needs to create a Registerable object by invoking the *createRegisterable()* operation on a RegisterableFactory object. The Registerable interface inherits the following methods from the trader Register interface: *export()*, *withdraw()*, *update()*, etc. These methods allow managing service offers.

The service's view

This view is defined by the Sharable interface, which should be supported by object servers. This interface defines two readonly attributes, that are respectively of types Monitorable and Registerable, and one operation called *getMonitorable()* that is invoked by the service in order to get the object reference of the Monitorable object associated to a well known object server. This object reference is used to get the current load value of the object server.

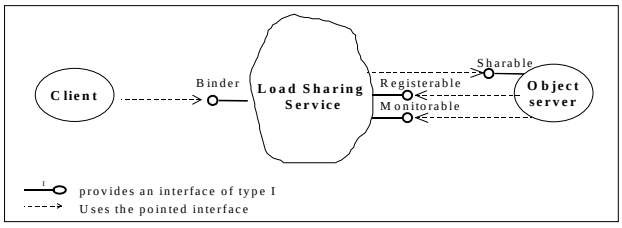

Figure 2 - Load Sharing Service interfaces

3.2. Load Sharing Service Scenarios

In this section, we present two scenarios showing how the interfaces of the load sharing service interact.

Client side scenario

Figure 3 shows how a client uses LSS in order to dynamically locate a suitable server that provides the required service while satisfying the minimal load and the client constraints.

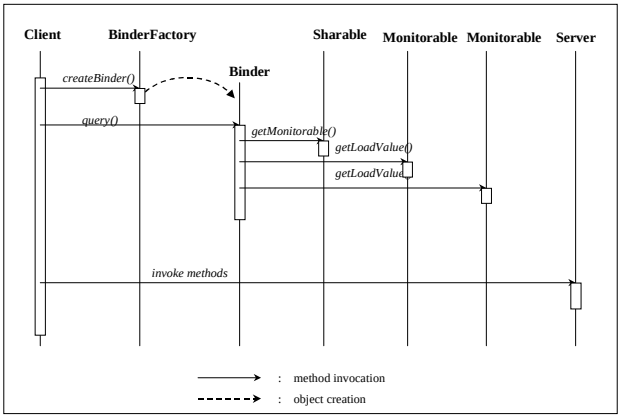

Figure 3 - Load Sharing Service - client side

The steps are as follows:

1. The client creates a Binder object by invoking the *createBinder()* operation on a BinderFactory interface.

2. The client formulates its request specifying the required service type, its constraints, and then, invokes the *query()* operation on the Binder object created in step 1. The result is a list of servers providing the required service. This list is sorted according to the current load of servers, and is obtained through the steps 3, 4, and 5.

3. The request is forwarded to the Lookup interface of the trader service by invoking its *query()* operation.

4. For each service offer of the list obtained in step 3, the Binder object invokes the *getMonitorable()* operation on the object referenced by the object reference of the service offer in order to get the object reference of the associated Monitorable object that monitors its load. The object reference of the service offer references a Sharable object (or an object server that inherits the Sharable interface).

5. The Binder invokes the *getLoadValue()* operation on each Monitorable objects obtained in step 4 in order to get the load of the servers discovered in step 2.

6. The client chooses the least loaded object server that can provide the service and invokes its methods to get the service if it has a stub for the server's interface.

Otherwise, it has to use the CORBA dynamic invocation service for building dynamic invocations.

One drawback of this scenario is that once the client has obtained the reference object of the current least loaded server, it can use it to make subsequent service requests to this server. Consequently, these requests may be serviced by servers which are not least loaded. To avoid this situation, the client has to implement a *smart proxy* that acts on its behalf to get the reference of the current least loaded server prior to sending a new service request. The smart proxy feature is provided only by few commercial ORBs such as Orbix from Iona Technologies [6].

Server side scenario

Figure 4 shows how an object server advertises its service offer and how its load is monitored. An object server should inherit the Sharable interface as shown in figure 1.

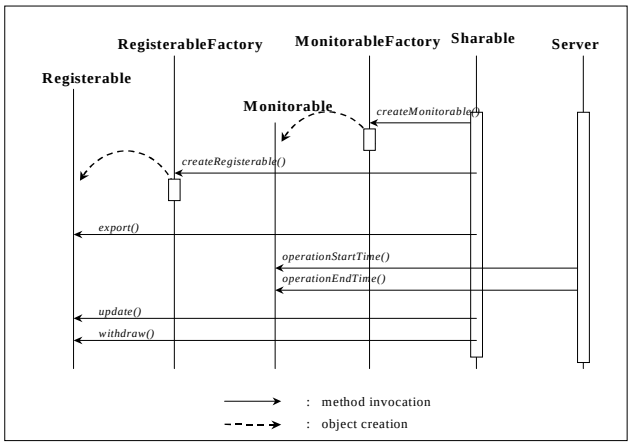

Figure 4 - Load Sharing Service - Server side

The steps are as follows:

1. The Sharable object (or an object server inheriting the Sharable interface) creates a Monitorable object, by invoking the *createMonitorable()* operation on a MonitorableFactory object, that monitors its load.

2. The Sharable object creates a Registerable object by invoking the *createRegisterable()* operation on a RegisterableFactory object. This allows the object server to advertise its service offer.

3. The object server exports its service offer by invoking the *export()* operation on the Registerable object created in step 2. This request is forwarded to the Register interface of the trader.

4. Once the object server methods are invoked by clients, the associated Monitorable object created in step 1 is notified of the arrival of a new request to the object server by invoking the *operationStartTime()* operation, and of the end of processing of the request by invoking the *operationEndTime()* operation. These two operations allow calculating the server busy time during a unit of time.

5. If the object server wishes to modify its service offer, it invokes the *update()* operation on the *Registrable* object created in step 3.

6. If the object server wishes to withdraw its service offer, it invokes the *withdraw()* operation on the Registerable object created in step 3.

4. EXPERIMENTATION AND RESULTS

The load sharing service has been implemented using the OrbixWeb [6] ORB and the OrbixTrader [8] CORBA Trading Object Service from Iona Technologies, and the Java programming language. The execution environment under which our design is implemented consists of two Sun Workstations running Solaris 2.5, which are connected by a 10Mbps Ethernet.

The goal of the experiments we have conducted is to evaluate the performance benefits of the LSS service. The performance metric we consider is the average response time of service requests issued by clients. The service type we have used is a bank service with operations for money deposit, money withdrawal, and balance consultation. Four servers named server1, server2, server3, and server4 respectively implement this service. We perform load distribution between these four servers using the following three methods for server selection.

1. *Least loaded server selection using LSS*: load is shared between servers using the LSS service. Client requests are sent to least loaded servers determined by the Binder objects. The load of a server is defined by the server utilization, i.e. the amount of time (in milliseconds) the server is busy handling requests during one period of time (1 second).

2. *Round-robin server selection*: the four servers are selected in a cyclical way, and client requests are sent to the selected servers.

3. *Random server selection*: the four servers are selected in a random way, and client requests are sent to the selected servers.

We measure the average response time for the same service request (get_balance()) in order to allow to accurately compare the three methods.

A set of similar clients of the bank service has been implemented. These clients generate service requests according to Poisson distributions with the same mean rate (λ=5). Requests are assigned to servers according to the three considered policies for server selection. For each

request, the elapsed time between request submission and the arrival of the result (response time) is measured. We consider situations with respectively 1, 2, 4, 6, 10, 14, and 20 simultaneous clients. All the client programs were executed on the same host. The communication costs are not considered in the present work.

Num. of clients	Least-loaded selection (LSS)		Random selection		Round-robin selection	
	Num. of req.	ART (ms)	Num. of req.	ART (ms)	Num. of req.	ART (ms)
1	3	18	5	16	4	16
2	15	18	10	15	9	15.5
4	27	20.25	28	18	22	21.75
6	30	21.5	34	26.17	29	24.5
10	73	25.2	53	34.3	50	29.3
14	87	25.21	86	42.64	89	44.64
20	131	31.15	122	50.9	105	59.05

Table 1 - Generated requests and average response times per number of clients

Table 1 lists the total number of generated requests and the associated average response time (ART) for each case of the above situations and for each server selection policy. Figure 5 shows the average response time as a function of the number of requests. At small loads, the three methods perform nearly equivalently. The slightly differences in average response time at small loads are principally due to experimental variations, since all the four servers of the experiment are under-loaded. The benefits of the LSS load sharing strategy increases with the number of requests. In contrast, random and round robin show increasingly worse performance as the number of requests increases. We think that this is because with LSS load is uniformly distributed among the four servers as it is illustrated in figure 6. Whereas, some servers may have excessive load with the random and the round robin methods.

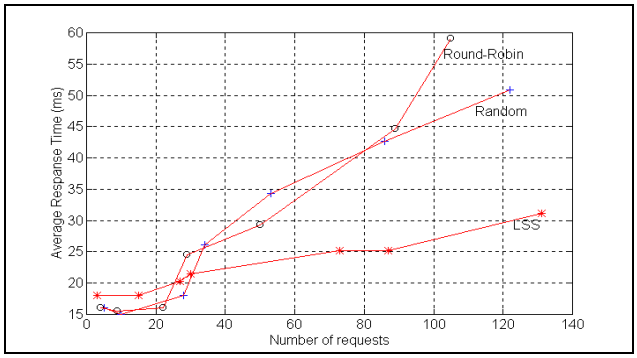

Figure 5 - Evolution of the average response time with the number of requests

During the experiment of LSS with 20 clients, we recorded the load of each server and the time at which is has been measured. Figure 6 shows the evolution of the load for each server. The load of the four servers follows nearly the same pattern. Differences mainly appear at the beginning of the experiment. We think that this is because clients are activated sequentially and because of the necessary operations to activate the servers and their associated objects (Monitorable and Registerable objects).

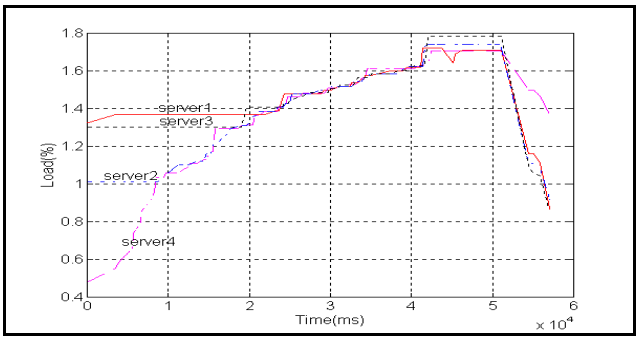

Figure 6 - Evolution of the servers' load with LSS for 20 simultaneous clients

These results show that with LSS the load is uniformly distributed among the four servers of the experiment, and that LSS outperforms the random and round robin methods.

5. RELATED WORK

Only few works have investigated the issues of dynamic load distribution and high availability in CORBA environments, and only few ORB vendors have announced that their products provide support for dynamic load distribution.

The LYDIA project [1] describes a centralized architecture for request assignment in CORBA environments. Load balancing decisions are taken by a load balancer, which cooperates with the naming service to perform request assignment. Monitoring is achieved through the instrumentation of clients and servers by modifying respectively their stubs and skeletons. This solution is not portable since stubs and skeletons are generated by an IDL compiler for a specific ORB. The LOCA project [14] investigates two dynamic load-distributing strategies for CORBA applications: request assignment and object migration. The request assignment approach in LOCA is to some extent similar to the approach we have used in LoDACE.

P. Felber et al [12] describe the design of a CORBA group communication service. S. Maffeis [13] also presents an object group design pattern for group communication and fault tolerance in CORBA distributed systems. Group communication is considered a powerful paradigm to support fault tolerance and high availability through replication. A service request sent to a group is executed by all the servers of the group in a transparent way for the client. The disadvantage of this paradigm lies in the fact that some servers of the group are likely to be overloaded especially when requests are of long duration.

Recently, certain ORBs vendors have announced that their products support load balancing. CORBAplus [4] offers to the developer four algorithms to balance the load among a set of servers: *byQueueSize*, *byProcessLoad*, *byExplicitsetting*, and *byRoundRobin*. The BEA ObjectBroker [2] uses the concept of "SmartMaps" that allows several policies for binding to servers. Orbix OTM [7] and the IBM Component Broker Connector [5] provides load distribution using the round-robin server selection method. As opposed to LSS, load sharing in all these products is not based on the current load of servers, except for CORBAplus, and is not portable since in most cases it relies on specific features of the product.

6. CONCLUSION

In this paper, we have described the issue of load sharing in CORBA environments. We have presented the design of a CORBA compliant load sharing service (LSS) which is based on a trading service and on load monitoring of servers. Experimentation results show that LSS allows the uniform load distribution among the servers. In addition, LSS outperforms the random and the round robin server selection methods, in terms of the average response time for client requests.

Below, we will discuss the LSS service in terms of scalability, availability, performance, and fault tolerance, and address future work. 1) *Scalability*: The design principles of LSS as presented in Section 2.1 allow for scalability. Object servers of a distributed application may be added to respond to the increasing demand of clients, and several servers may run on the same host. 2) *Availability and performance*: sharing the load between peer servers providing the same service type increases the availability of servers and reduces the response time of client requests. Servers providing inadequate response times may be identified and additional instances of these servers may be created either on the same host or on other hosts of the system. 3) *Fault tolerance*: using LSS, partial failure of some servers won't prevent the service from being provided. When the failure of a server is detected, the client gets the object reference of another suitable server from its Binder object. However, the failed server's state is lost. Sophisticated fault tolerance techniques, such as checkpointing and group communications, may be integrated to the LSS service to provide a fully fault tolerant service.

As future work, we intend to extend LSS to support load-sharing techniques such as sender-initiated and receiver-initiated policies. Moreover, further extensive tests including the above mentioned extensions will be made. Finally, the formal analysis of the LSS method will make it possible to predict the performance and to validate the experimental results.

REFERENCES

[1] B. Schiemann, "Specification of IDL Mechanisms for Load Balancing (First Draft), ESPRIT III P8144, LYDIA/WP.4/T.4.2/D6, 1996.

[2] BEA System Inc., "Presentation de BEA ObjectBroker", http://www.beasys.fr/produits/objectbroker.htm.

[3] E. Badidi , R. K. Keller, P. G. Kropf, and V. Van Dongen, "LoDACE: une architecture de partage de charge dans les systèmes distribués objets", Proc. of the Colloque Int. sur les NOuvelles TEchnologies de la Repartition, Montreal, QC, Canada, pp. 281-296, 1998.

[4] ExperSoft Coporation, "CorbaPlus: The Expersoft Difference: Per Invocation Load Balancing ", http://www.expersoft.com/Products/TechAdv/Load_balance.htm.

[5] IBM, Intern. Technical Support Org., Austin Center, "IBM Component Broker Connector, Overview", SG24-2022-01, November 1997.

[6] Iona Technologies Ltd., OrbixWeb programming guide, 1996.

[7] Iona Technologies PLC, "The Orbix Object Transaction Monitor (OTM)", White paper. http://www.iona.com/support/whitepapers/orbixotm/otm_wp.html

[8] Iona Technologies PLC, OrbixTrader Programmer's Guide and Reference, 1997.

[9] Microsoft Corporation, "Microsoft Windows NT Server, DCOM Technical Overview ", White Paper, 1996.

[10] OMG, CORBAservices: Common Object Services Specification, Updated version, 1997.

[11] OMG, The Common Object Request Broker: Architecture and Specification, Rev.2.1, 1997.

[12] P. Felber, B. Garbinato, and R. Guerraoui, "The design of a CORBA group communication service", IEEE Int. Conf. on Reliable Distributed Systems, pp. 150-159, Niagara, Canada, 1996.

[13] S. Maffeis, "The Object Group Design Pattern", Proc. of the USENIX Conference on Object-Oriented Technologies, Toronto, Canada, 1996.

[14] T. Schnekenburger and G. Rackl, "Implementing Dynamic Load Distribution Strategies with Orbix", Proc. Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97), Las Vegas, Nevada, Volume II, pp. 996-1005, 1997.