A Scalable Architecture for Parallel CORBA-based Applications

MARKUS ALEKSY
Department of Information Systems III
University of Mannheim
Schloß (L 5,6)
68131 Mannheim, Germany
+49 621 181 1488
aleksy@wifo3.uni-mannheim.de

MARTIN SCHADER
Department of Information Systems III
University of Mannheim
Schloß (L 5,6)
68131 Mannheim, Germany
+49 621 181 1639
mscha@wifo3.uni-mannheim.de

ABSTRACT

In this paper we present a scalable architecture for parallel CORBA-based systems. It extends CORBA's basic synchronous communication model and enables enhanced flexibility in programming applications where a client makes requests to a multitude of servers and expects one single result as a response. Individual requests may be distributed depending on system load and may involve several layers of distribution. The architecture we discuss was tested with a CORBA-based implementation.

KEY WORDS:

CORBA, group communication, load balancing

1. INTRODUCTION

In the era of the internet, scalable software architectures gain in significance. This is due to the fact that software developers have no knowledge about the number of users that will access their application. The application must be able to comply with the growth of the company and its customers. Even if, in the startup phase, only some few staff members or clients use the application, the number of accesses might multiply within a very short time. Especially in the internet sector, *one-to-many-to-one*-architectures (1:N:1) gain increasing importance. Here, the request of a client – called master in the following – is delivered to an arbitrary number of servers – called workers in the following (1:N). The workers execute their task and deliver the response to the master who has to aggregate and evaluate the individual results (N:1). This 1:N:1 archi-tecture should not be mistaken for the standard three tier architecture with clients, application server, and database backend.

If we analyze the processing of such a call in more detail the necessary communication techniques become apparent:

- group communication and
- asynchronicity.

Group communication is needed to dispatch the request to the workers that will execute it in parallel. Asynchronicity is required since the request must be distributed asynchronously and the workers' responses will arrive at different times.

The basic communication model propagated by CORBA [1] is synchronous and blocking. No standards on group communication are specified. Programmers, therefore, face the problem that the techniques needed for developing a 1:N:1 application are only insufficiently standardized. In [2] we have previously described CORBA-based solutions for asynchronous communication and discussed their suitability. Further aspects of asynchronous communication techniques in CORBA applications can be found in [3] and [4].

Some examples of 1:N:1 applications are:

- a meta search engine,
- an electronic broker,
- cluster management or
- execution of a parallel program.

In the first example the user communicates a request to a meta search engine which forwards this request to several independent search engine services (1:N). These further transmit the request to the corresponding search engines, e.g., `www.lycos.com`, `www.altavista.com`, or `www.google.com`. The answers of the search engines will come in at different time points and have to be aggregated and evaluated (N:1).

In the second example, an auctioneer sends an initial offer to the participating bidders (1:N). Bidders can submit their individual bids in a given time period or at a specific time point. The bids then have to be aggregated and evaluated (N:1).

The administration of computer clusters also requires a 1:N:1 architecture. If state information on the different nodes, e.g., concerning CPU load or drive capacity is needed the monitoring tool has to send requests to each node (1:N) and then must evaluate the individual node information (N:1).

The last case is the most complex example. Here, a master sends different input data to a multitude of workers. Typically, original data have to be distributed among the workers (1:N). The individual workers process their input and return the results to the master (N:1). In order to obtain the final result the master must evaluate the partial results of the participating workers.

In all four cases we discussed, the same pattern emerges: distribution of the original request to the workers and aggregation of the respective results has to be resolved. Providing a transparent solution to this task would substantially simplify programming of master and workers since parallel data processing would be hidden from both. Developers could avoid the employment of potentially error-prone multithreading techniques and of complex synchronization and coordination mechanisms.

This problem – the transparent distribution and aggregation of requests including necessary synchronization and co-ordination tasks – can be solved by applying the architect-ure we suggest.

2. DESIGN OF THE SUGGESTED ARCHITECTURE

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

- Portability. Only minimal vendor- or plattform-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 our architecture easily to the characteristics of the respective application.

3. THE CORE ARCHITECTURE

The core architecture is based on two CORBA services:

- an Object Group Service (OGS) and
- a Join Service (JS).

The general idea of a CORBA-based OGS originates in the work of Felber [5] who has suggested this approach for implementing replication scenarios. Our design and the corresponding implementation, however, aim at supporting parallel invocation of CORBA operations, i.e., a request issued by the master is concurrently propagated to an arbitrary number of workers which are able to process the data they receive independently (1:N). In addition to distributing the complete data set to all group members, our solution supports further data dispatching strategies such as `PER_ELEMENT`, `SAME_SIZE`, or `DIFFERENT_SIZE`. This reduces network load and, on the other hand, data distribution enables parallel processing. The number of transferred bytes is maximally twice the number of bytes in the original data. Should within-group communication be based on CORBA's Event Service [6] and not on an OGS, all registered Event Consumers – who would, in this scenario, correspond to the workers – will receive the complete data set, whether they need it or not. Thus, network load increases with the number of workers and amounts to `(n+1)m` bytes if `m` is the size of the transferred data and `n` is the number of Event Consumers. A detailed description of the OGS and the provided data dispatching strategies can be found in [7].

The design of our OGS allows the distribution of the original data with no restrictions on the data types or data structures needed. Developers only have to ensure that data is organized as an unbounded CORBA sequence, i.e., a vector with variable length, in case that a different strategy than simply passing the complete data to all workers is to be applied. The data elements in the sequence can be of any simple or complex structure. Basic types, as well as complex types containing other types such as basic types, arrays, sequences, and (possibly layered) structures are permitted. At runtime the OGS dynamically determines the types of data contained in the CORBA sequences and copies the data into new subsequences of the same type. The number of subsequences that have to be created in that way depends on the number of workers and the number of data sets to be dispatched. A positive aspect of this approach is the possibility to distribute sequences of data whose types were not known when the OGS was developed. Thus, the OGS is capable of covering a broad field of current and future applications. The price that has to be paid for this flexibility is a certain loss in performance since marshalling and unmarshalling of the type `any` takes up more time than that of basic types.

In order to be able to use the OGS, an OGS client has to implement the interface `Master`. This interface defines only a single operation `receive()`, that receives a message containing the collection of results of the different workers via the JS. An OGS server, on the other hand, must implement the interface `Worker` with its operation `send()`. `send()` not only passes information on the operation to be executed but is also provided with the Interoperable Object Reference (IOR) of the JS. The IOR is necessary to enable passing of the result back to the master via a callback. In a first step, the master sends a message and its IOR to a certain group by calling `send()`. In the second step, the group dispatches the message to its members (the workers) and informs the JS on the workers' results to be expected. The JS then receives the results of the workers and, finally, calls

`receive()` to combine theses results to a single end result and communicate it to the master.

The IDL interface `Group` serves the purpose of propagating a call issued by a master to all members of the group. It provides functionality for adding and removing workers to and from the group. In order to administer several groups of workers we also defined a `GroupManager` interface. It contains operations for generating, listing, finding, and deleting groups.

The JS is the counterpart of the OGS. Its task is to collect and combine the results of the requests issued to the workers. This end result is, finally, sent back to the master. The service also supports two strategies:

- `COMPLETE` and
- `PARTIAL`.

The `COMPLETE` strategy has the effect that the JS will wait for the results of all the workers involved. This implies that the crash of a single worker during its work will cause the JS to wait infinitely.

Using the `PARTIAL` strategy this drawback can be avoided. Here, a timeout value has to be specified. In this case, the JS at most waits for the specified time and, subsequently, reports back the results of the workers that have already delivered. If all workers send their results in time the JS can report the complete end result before timeout.

A detailed description of the OGS/JS architecture is given in [7]. The cooperation of the components of the core architecture is depicted in Figure 1.

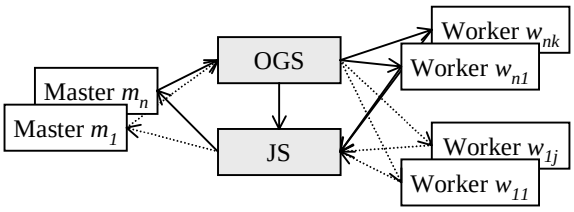

Figure 1: Design of the core architecture

In the case of a small number of masters and workers the OGS can be started with the option `-use-internal-JS`. The OGS then uses its internal JS for combining the results of the workers. This internal JS is also employed if an externally started JS cannot be found. In that case the activities of the OGS follow the Master/Slave pattern decribed in [8].

4. SCALABILITY OF THE CORE ARCHITECTURE

The core architecture described in the previous section enables 1:N:1 communication, but with increasing numbers of masters and workers or with increasing data sizes both services can become bottlenecks. To overcome this potential problem it is possible to start more than one instance of OGS and JS. Accessing a Naming Service [9] or a Trading Service [10], a master can obtain IORs of different OGSs and then decide to which OGS a request should be sent. Figure 2 illustrates this scenario.

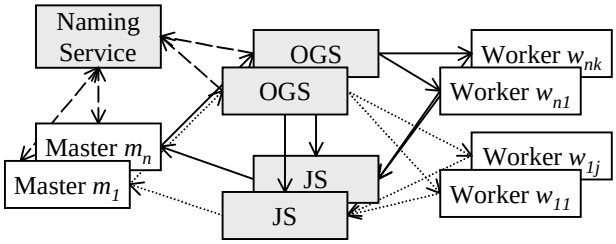

Figure 2: Scalability of the core architecture

Additionally, the OGS can direct groups of workers to deliver their results to different JS instances. In our implementation the selection of the responsible JS is based on a round robin balancing strategy. Within the same application we can start OGSs that employ a single internal or external JS as well as OGSs that use several different JS instances concurrently.

The problem solution described so far, however, is not satisfactory in all cases since it offers no indication on the load of the OGS and its workers. In the following sections we will, therefore, discuss an extension of the core architecture that can improve scalability significantly by applying a load-dependent distribution scheme.

5. EXTENSIONS OF THE CORE ARCHITECTURE

To obtain an efficient load distribution among OGSs, JSs, and their workers we added a Load-Balancing Service to the core architecture.

5.1. Load-Balancing Service (LBS)

Like all other CORBA services, the LBS [11] was conceived as an independent central component. This decision involves several advantages but also some disadvantages. On the positive side we can mention:

- information gathered on service and worker 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 counterproductive or unfavorable decisions,
- with a centrally controled assignment of new requests a rather uneven distribution of tasks can usually be prevented.

Some of the disadvantages should also be noted:

- recording information on service and worker load additionally burdens the LBS,
- as a central component, the LBS is not or not fully scalable,

- as a central component, the LBS constitutes a single point of failure; its breakdown causes the failure of all connected applications.

The first criticism can be met if we delegate the recording of load information to the individual servers. Here, different workers as well as services whose load needs to be recorded, e.g., the OGS, are seen as servers. Assigning load information retrieval to the servers somewhat slows down their performance but it will considerably relieve the LBS since it is a central component.

The next two disadvantages can be overcome if – at run-time of the applications – we start several LBS instances and register them with the Name Service or Trading Service. This approach, however, increases expenses on the side of the master who now has to obtain and examine different names before submitting a request.

The LBS at present consists of two main components:

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

The `ServantMonitor` is responsible for gathering load data. It provides a standard query interface (`Monitor-DataPool`) which the `Balancer` and the masters can access. Determination of the load parameters can be based on the total load of a server or individually on each CORBA object. In that context (information gathering) 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. Again, workers as well as services like the OGS can act as servers. The second task of the `Balancer` is to work as a *Locator* that sends a complete list of eligible servers to the caller without making any scheduling decision. 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 (a master, e.g.) 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 when the requests do not change the server's state since, otherwise, inconsistencies might occurr. Should the LBS component, nevertheless, become a bottleneck we can construct more than one `Balancer` instance.

The `Balancer` supports a variety of 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 while their resource consumption is limited.

The main static algorithms implemented in the LBS are:

- `RANDOM`. Here, the server is randomly selected.
- `ROUND_ROBIN`. One server after another is selected for processing a client request.
- `LONGEST_UNUSED`. The server that was not used longest is selected.

Examples of dynamic algorithms supported by the LBS are:

- `MAX_IDLE`. The server that was idle longest is selected.
- `HIGHEST_IDLE_PERCENTAGE`. The server that was idle most frequently is selected.
- `FEWEST_REQUEST_PER_SECOND`. The server that processed the fewest number of requests per second is selected.
- `SHORTEST_REQUEST_AVG`. The server with the smallest average processing time is selected.
- `SHORTEST_REQUEST_MIN`. The server with the smallest minimal processing time is selected.
- `SHORTEST_REQUEST_MAX`. The server with the smallest maximal processing time is selected.
- `FEWEST_CLIENTS`. The server with the smallest number of completed results is selected.
- `BEST_MIX_INDEX`. The server 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 LBS.

Other approaches in the area of CORBA-based load monitoring and load balancing can be found in [12] and [13].

5.2. The LBS from the Perspective of the Master

As we described above, a master can instruct 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 server load dynamically, 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 master itself determines the load of the servers, selects the best suited server or servers and sends its request.

5.3. Programming Example

In a first step, the master has to resolve the IOR of the LBS. The necessary methods are supplied by the class `ClientUtil`. Through a lookup at the Name Service it tries to obtain access to the LBS. In the next step, the master must specify the desired load balancing strategy. In this example we select the static `ROUND_ROBIN` algorithm. Then, calling `import()`, the master receives the desired reference to a worker. The following Java code snippet will illustrate this procedure.

```
ORB orb = ORB.init();

org.omg.CORBA.Object servantRef =
  ClientUtil.import(
    ClientUtil.resolveBalancer(
      orb, "LB1", ""),
    WorkerHelper.id(),
    new Policy[] {
      Client.create_policy(orb,
        StaticSelectionCriterion.
        ROUND_ROBIN)
    },
    new PolicySeqHolder()
  );
```

It can be seen that a master using the LBS approach needs only few extensions compared to its unbalanced equivalent. Modifications on the worker's side are also minimal. Therefore, existing OGS/JS applications can be easily migrated to a load balanced version.

5.4. Extensions of the 1:N:1 Architecture

Depending on the number of active masters, workers, OGSs, and JSs and considering actual load as well as size or complexity of the application, the best suited number of LBSs will vary. To determine the number of LBSs needed, different scenarios are appropriate:

- Load Balancing on the service level.
  Following this strategy, only OGS server load is measured. In that case the LBS has to forward incoming requests to the OGS which is minimally loaded. Figure 3 illustrates this approach. Distributing load among several OGS instances only results in limited improvements of load balance since load of the existing groups and their workers is not taken into consideration.

- Load Balancing on the group level.
  This approach seems more promising. As discussed earlier, a client can select the appropriate group according to its preferences. Only the preferred server type has to be specified and the result of the request will now be a group. This technique has a further advantage: transparency of location. Selection of a group from different OGS instances is possible and the client will not detect that the currently selected group is registered with a different OGS than the one the client received from the last request.

  Detecting the best worker would also be possible but, usually, offers no improvement. In that case, the result of a request to this worker will be delivered faster in comparison to other group members. The performance of the complete group might, however, be considerably worse than that of other groups. The reason is that overall group performance does not depend on a single worker but is determined by all workers in the group. The task of measuring overall group load can therefore only be accepted by the group itself which has all relevant information needed. To solve this problem, the group has to register with the LBS and is treated as a single server object.

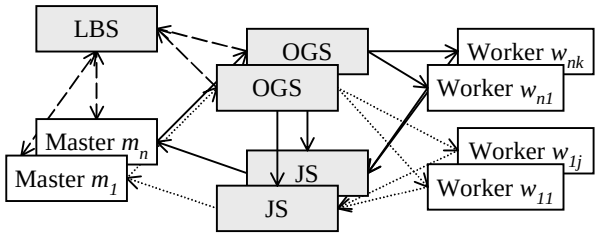

Figure 3: Extended architecture for load balancing

6. CRITICAL REVIEW

Our architecture supports CORBA-based distributed parallel processing of data independently of the used programming language. Concurrency of processing on several workers is transparent to the user, therefore, the master as well as the workers can be implemented as simple sequential programs. Especially developers who implement their applications in a programming language that does not support multithreading satisfactory can benefit from our approach and develop parallel applications in a simple way. Moreover, the approach is helpful for developers with limited experience in parallel programming. They do not have to consider data dispatching or synchronization problems since the OGS can be activated through a single simple operation call and the result of the JS is also obtained by a single call. Thus, developers can avoid employing potentially error-prone techniques or complex synchronization and coordination mechanisms.

With the help of the data distribution strategies implemented in the OGS network load can be perceivably reduced. Furthermore, users are allowed to select the strategy best suited for their application. The implemented algorithms can be reused directly but they also may be easily modified.

A load dependent dispatching of requests is possible through the LBS extension of the core architecture. Analogies to the Trading Service will shorten the learning phase for developers that have experience with that CORBA service.

When relying on the LBS, developers can influence the selection of the resulting service or worker through different criteria. Static or dynamic algorithms can be applied. If these criteria are not sufficient one or more workers can be requested for exclusive use. It is also possible to request a list of references of all available workers and to select one or more workers from that list.

The extensive means for configuration enables users to flexibly apply the proposed architecture to a large variety of problem domains. Especially in the internet domain where *one-to-many-to-one*-architectures play an important role, application development can be supported.

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.

7. REFERENCES

[1] OMG, CORBA/IIOP 2.4 Specification, OMG Technical Document Number 00-10-01, 2000, ftp://ftp.omg.org/pub/docs/formal/00-10-01.pdf

[2] M. Aleksy, A. Korthaus, Implementation Techniques and an Object Group Service for CORBA-Based Applications in the Field of Parallel Processing, *Proc. Seventh International Conference on Parallel and Distributed Systems*, Iwate, Japan, 2000, 65-72

[3] D. C. Schmidt, S. Vinoski, An Introduction to CORBA Messaging", *C++ Report, 10*(10), 1998

[4] D. C. Schmidt, S. Vinoski, Programming Asynchronous Method Invocations with CORBA Messaging, *C++ Report*, *11*(2) , 1999

[5] P. Felber, B. Garbinato, R. Guerraoui, The design of a CORBA group communication service, *Proc. of the 15th IEEE Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, USA*, 1996, ftp://ftp-lse.epfl.ch/pub/felber/papers/SRDS-96.ps

[6] OMG, Event Service Specification, *OMG Technical Document Number 00-06-15*, 2000, ftp://www.omg.org/pubs/docs/formal/00-06-15.pdf

[7] M. Aleksy, A. Korthaus, A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming, *Proc. of the International Symposium Software Engineering for Parallel and Distributed Systems*, Limerick, Ireland, 2000, 123-134

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns – Pattern-Oriented Software Architecture* (John Wiley & Sons, Chichester, 1996)

[9] OMG, Naming Service Specification, *OMG Technical Document Number 00-06-19*, 2000, ftp://ftp.omg.org/pub/docs/formal/00-06-19.pdf

[10] OMG, Trading Object Service Specification, OMG *Technical Document Number 00-06-27*, 2000, ftp://ftp.omg.org/pub/docs/ formal/00-06-27.pdf

[11] M. Hoffmann, Entwurf und Implementierung eines CORBA-basierten Load-Balancing Service, *Master Thesis, Department of Management Information Systems III*, University of Mannheim, 2000

[12] K. Geihs, C. Gebauer, Load Monitor LM – Ein CORBA-basiertes Werkzeug zur Lastbestimmung in heterogenen verteilten Systemen, *9. ITG/GI-Fachtagung MMB'97, "Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen"*, TU Bergakademie Freiberg/Sachsen, 1997

[13] T. Barth, G. Flender, B. Freisleben, F. Thilo: Load Distribution in a CORBA Environment, *Proc. International Symposium on Distributed Objects and Applications*, Edinburgh, Scotland, 1999, 158-166