CORBA Based Runtime Support for Load Distribution and Fault Tolerance

Thomas Barth, Gerd Flender, Bernd Freisleben, Manfred Grauer, and Frank Thilo

University of Siegen, Hölderlinstr.3, D–57068 Siegen, Germany
{barth, grauer, thilo}@fb5.uni-siegen.de,
{freisleb, plgerd}@informatik.uni-siegen.de

Abstract. Parallel scientific computing in a distributed computing environment based on CORBA requires additional services not (yet) included in the CORBA specification: load distribution and fault tolerance. Both of them are essential for long running applications with high computational demands as in the case of computational engineering applications. The proposed approach for providing these services is based on integrating load distribution into the CORBA naming service which in turn relies on information provided by the underlying WINNER resource management system developed for typical networked Unix workstation environments. The support of fault tolerance is based on error detection and backward recovery by introducing proxy objects which manage checkpointing and restart of services in case of failures. A prototypical implementation of the complete system is presented, and performance results obtained for the parallel optimization of a mathematical benchmark function are discussed.

1 Introduction

Object–oriented software architectures for distributed computing environments based on the *Common Object Request Broker Architecture* (CORBA) have started to offer real-life production solutions to interoperability problems in various business applications, most notably in the banking and financial areas. In contrast, most of todays applications for distributed scientific computing traditionally use message passing as the means for communication between processes residing on the nodes of a dedicated parallel multiprocessor architecture. Message passing is strongly related to the way communication is realized in parallel hardware and is particularly adequate for applications where data is frequently exchanged between nodes. Examples are data–parallel algorithms for complex numerical computations, such as in computational fluid dynamics where essentially algebraic operations on large matrices are performed.

The advent of networks of workstations (NOW) as cost effective means for parallel computing and the advances of object-oriented software engineering methods have fostered efforts to develop distributed object-oriented software infrastructures for performing scientific computing applications on NOWs and also over the WWW [7]. Other computationally intensive engineering applications with different communication requirements, such as simulations and/or multidisciplinary optimization (MDO) problems [3] [5] typically arising in the automotive or aerospace industry, have even strengthened the need for a suitable infrastructure for distributed/parallel computing.

Two essential features of such an infrastructure are load distribution and a certain level of fault tolerance. Load distribution improves the effectiveness of the given resources, resulting in reduced computation times. Fault tolerance is important especially for long–running engineering applications like MDO software systems. It is obviously crucial to provide mechanisms to prevent the whole computation from failing due to a single error on the server side.

In this paper, CORBA based runtime support for parallel applications is presented. This support encompasses load distribution as well as fault tolerance for parallel applications using CORBA as communication middleware.

2 Integrating Load Distribution into CORBA

In general, CORBA applications consist of a set of clients (applications objects) requesting a set of services. These services can either be other application objects within a distributed application, or commonly available services (object services) providing e.g. name resolution (naming service) or object persistence (persistence service). There are different approaches to integrate load distribution functionality into a CORBA environment:

– Implementation of an explicit service (e.g. a "trader", [12]) which returns an object reference for the requested service on an available host (centralized load distribution strategy) or references for all available service objects. In the latter case, the client has to evaluate the load information for all of the returned references and has to make a selection by itself (decentralized load distribution strategy).
– Integrating the load distribution mechanism into the ORB itself, e.g. by replacing the default locator by a locator with an integrated load distribution strategy [6] or using an IDL–level approach [13]

The drawbacks of these approaches are either that the source code of clients has to be changed (as in the first approach) or that load distribution depends on a specific ORB implementation or IDL compiler and can thus not be utilized when other ORBs are used (as in the second approach). To integrate load distribution transparently into a CORBA environment, our proposal is based on integrating it into the naming service. This ensures transparency for the client side and allows the reuse of the load distribution naming service in any other CORBA–compliant ORB implementation. In almost every CORBA–based implementation the naming service is utilized. In the case of applications which do not make use of the naming service, it would be useful to implement load distribution as an explicit service.

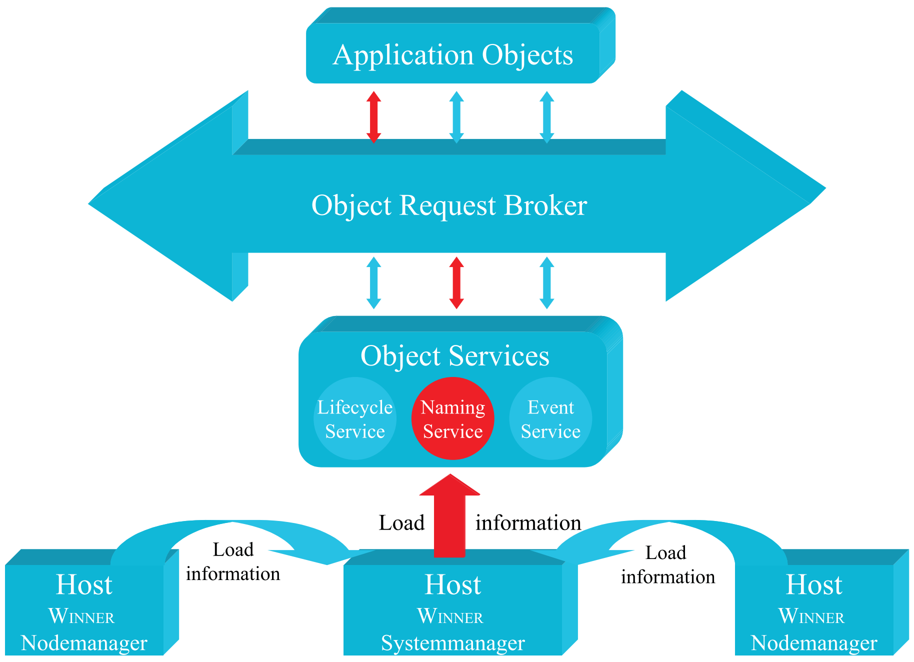

Fig. 1. *Schema for the integration of load distribution in a naming service.*

Our concept is illustrated in Fig. 1; it relies on the WINNER resource management system [1] [2]. Basically, WINNER provides load distribution services for a network of Unix workstations. Its components of interest here are the central system manager and the node managers. There is one node manager on each participating workstation, periodically measuring the node's performance and system load, i.e. data like CPU utilization which is collected by the host operating system. This data is sent to the system manager, which has functionality to determine the machine with the currently best performance. Requests from application objects to the naming service are resolved using this load information for the selection of an appropriate server.

The naming service is not an integral part of a CORBA ORB but is always implemented as a CORBA service. The OMG specifies the interface of a naming service without making assumptions about implementation details of the service. Therefore, every ORB can interoperate with a new naming service as long as it complies to the OMG specification.

3 Runtime Support for Fault Tolerance in CORBA Based Systems

The CORBA specification as well as the Common Object Services Specification offer no adequate level of fault tolerance yet. Due to the need for fault tolerance in more complex distributed systems, various approaches were developed. The Piranha system [8] for example, is based on an ORB supporting object groups, failure detection etc. Using these facilities provided by the ORB, Piranha is implemented as an CORBA Object Service for monitoring distributed applications and managing fault tolerance via active or passive replication. The major drawback of Piranha is its dependency on non–standard ORB features like object groups. Another approach avoids this drawback by complying completely to the CORBA standard: IGOR (**I**nteractive–**G**roup **O**bject–**R**eplication) [9] realizes fault tolerance also by managing groups of objects providing redundant services. In contrast to the Piranha system, IGOR is portable and interoperable with today's ORB implementations. Lately, there is also a proposal for the integration of redundancy, fault detection and recovery into the CORBA standard [10].

Unlike the previously mentioned approaches, our concept is not based on replicated services in object groups but on the integration of checkpointing and restarting functionality only. Especially for applications with a maximum degree of parallelism (e.g. scalable optimization algorithms) it is not desirable to use a large amount of the computational resources (i.e. hosts in the network) exclusively for availability purposes as in the case of active replication. Thus, in the case of parallel, long running applications it is a good compromise to restrict fault tolerance to checkpointing and restarting. Similar to the concept of passive replication, frequently (i.e. after each method call on the server side) generated checkpoints are used to restart a failed service.

Currently, the only way to detect an error on the client side of a CORBA application is the exception `CORBA::COMM_FAILURE` thrown when a CORBA client tries to call a service which is not available anymore (e.g. due to a network failure, a crashed server process or machine). Using the concepts for the naming service already described, it is possible to request a new reference to a service if a call to a server object fails. This approach is sufficient for services without an internal state. In the more general case of services depending on an internal state of the server object, it is inevitable to (a) save the state (checkpoint) of the server object e.g. after each successful call to a server's method and (b) have the opportunity to restore this state in a newly created server object.

We evaluated the following alternatives to integrate checkpointing and restarting functionality on the client side assuming that the service object provides a method to create a checkpoint for restarting the service if an error occurs: (a) modification of the client–side code to handle the `CORBA::COMM_FAILURE` exception and to restart a service, (b) extending the client–side stub code generated by the IDL–compiler with exception handling etc., and (c) introduction of proxy–classes derived from the stub classes on the client–side.

The major drawback of alternative (a) is the amount of code to be inserted on the client side: every single call from a client to a method of the server must at first get a checkpoint from the server, then handle the exception, and start a new server (using the checkpoint) in case of a failure. It would be useful if the automatically generated stub code comprises this code as in alternative (b). But this means changing the IDL–compiler itself, and thus this solution would be specific for a certain CORBA implementation providing its own IDL–compiler. Alternative (c) is a compromise between the amount of modifications to be made on the client side and the targeted platform independence of the concept: the modifications on the client side are limited to the use of a proxy class instead of the stub class. This proxy class is derived from the stub class and therefore provides all of the methods of the stub class. The additional methods handle the creation of a checkpoint and the restoring of an object's state according to a checkpoint. If a class offers this functionality for checkpointing and restoring a certain internal state it is in principle possible to migrate a service from host to another one not only when an error occured but also due to a changing load situation on a host.

With the current implementation, the proxy class for each service class has to be implemented manually. This could be easily automated by parsing the class definition. For each method, code to call the parent class (the stub) method along with exception handling code and a call to the server object's checkpoint and restore functions would have to be generated.

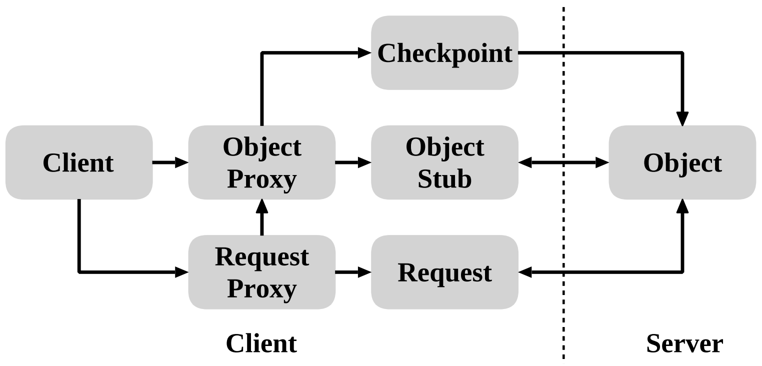

Fig. 2. *Scheme of client, server, proxy objects and their call relationship.*

As a proof of concept, a simple service for storing checkpointing data has been implemented. It simply provides functions to store/retrieve arbitrary values to the server object. No real persistency like storing checkpoints on disk media has been implemented, yet. Furthermore, the current implementation is rather inefficient. In addition to transparent synchronous method calls, CORBA provides asynchronous method invocations via DII (Dynamic Invocation Interface). When a client wants to utilize DII, it does not call the server object's methods directly, but uses so-called *request* objects instead. These request objects offer methods to asynchronously initiate methods of the server object and fetch the corresponding results at a later time. To enable fault tolerance in this case, *request proxies* are used just like the object proxies. The relationship between the described objects is shown in Fig. 2.

4 Experimental Results

To investigate the benefits of an integrated load distribution mechanism in CORBA, a test case from mathematical optimization was taken. The well known Rosenbrock test function [14] is widely used for benchmarking optimization algorithms because of its special mathematical properties. In our experiments, the function is only used to demonstrate the benefits of an adequate placement of computationally expensive processes on nodes of a NOW. It is not intended to present a new approach to the solution of the benchmark problem. To compute the function in parallel, a decomposed formulation of the Rosenbrock function has been taken. In the decomposed formulation, several (sub-)problems with a smaller dimension than the original n–dimensional problem are solved by workers, and the subproblems are then combined for the solution of the original problem in a manager.

In Fig. 3, the results of the different test scenarios are compared. All test cases were computed using multiple instances of a sequential implementation of the Complex Box algorithm [4] on a network of 10 workstations. The ORB used was omniORB 2.7.1 [11]. For the comparison of the different implementations of the naming service, a background load was generated on 0, 2, 4, 6 or 8 hosts. The

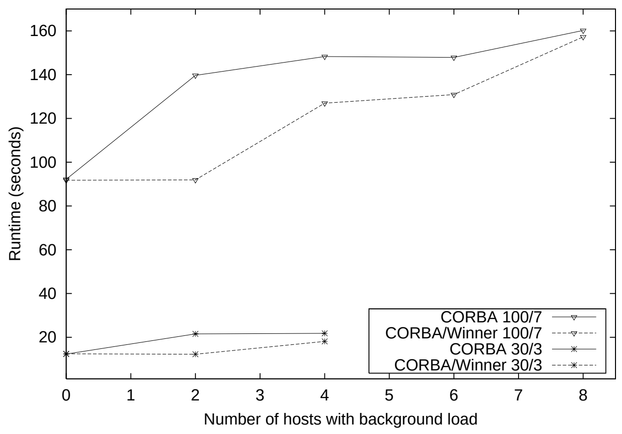

Fig. 3. *Different test cases of a decomposed 30– and 100–dimensional Rosenbrock function with 3 and 7 worker problems under different load situations.*

two lower curves show the computation times for a 30–dimensional Rosenbrock function with 3 worker problems (problem dimension 10, 9 and 9) and a 2–dimensional manager problem. In this scenario, 6 workstations were available for the 4 processes. The effect of load distribution is obvious when 2 hosts had background load. The selection of hosts with the new naming service avoided these hosts and hence the computation time was the same as in the case without background load. The two upper curves compare the computation times for a 100–dimensional Rosenbrock function with 7 worker problems. With increasing background, load the advantage diminishes because both implementations of the naming services are forced to select services on hosts with background load. To summarize, the benefit of load distribution for the test cases mentioned above can be estimated by ca. 40% runtime reduction in the best case. Even in the worst case it yields at least the same results as the unmodified naming service. The mathematical properties of the test cases as mentioned above result in an average reduction of computation time of about 15%.

Providing fault tolerance by proxy classes introduces an additional level of indirection. Additionally, storing the state of the server objects upon each method invocation causes some overhead. To quantify to what extent this overhead affects application runtimes, the above experiment has been repeated, this time using fault tolerant proxy classes. In Table 1 computation times for a 100–dimensional Rosenbrock problem are shown for the proxy and non–proxy case, respectively. The measurements have been conducted for different numbers of iterations of the worker's algorithm. The increasing number of iterations results in longer runtimes of the worker problems because it is a stopping criterion of the algorithm. Table 1 demonstrates that fault tolerance comes at quite a cost in this scenario. In the worst case, the application runtime using proxy objects is more than three times that of the plain version. Because the overhead is constant for each method call, the relative slowdown is lower the more time is spent in the called method. It is important to remark that when using real life engineering applications, most method calls will take orders of magnitude longer to finish. Additionally, the checkpoint storage class has not been optimized for speed in any way as the current implementation is merely a proof of concept.

Table 1. Runtimes for a 100–dimensional Rosenbrock function with 7 worker problems and a varying number of worker iterations.

Iterations	Runtime without proxy [s]	Runtime with proxy [s]	Overhead [%]
10,000	92	309	235.9
20,000	165	376	127.8
30,000	232	445	91.8
40,000	299	505	68.9
50,000	383	594	55.1

5 Conclusions

The design and implementation of a CORBA naming service providing load distribution and basic fault tolerance services based on proxy objects was presented. These services are essential for long–running computational engineering applications in distributed computing environments. Experiments demonstrated the feasibility of both concepts. Areas of future work are: (a) improving, optimizing, and stabilizing the prototype implementation of the proposed CORBA load distribution and fault tolerance services, (b) evaluating its benefits in real-life engineering MDO applications, and (c) extending the WINNER load measurement and process placement features for wide-area networks to enable CORBA based distributed/parallel meta-computing over the WWW. Additionally, the proposed extensions to the CORBA specification concerning redundancy, fault detection and recovery must be evaluated.

References

1. Arndt, O., Freisleben, B., Kielmann, T., Thilo, F., Scheduling Parallel Applications in Networks of Mixed Uniprocessor/Multiprocessor Workstations, Proc. Parallel and Distributed Computing Systems (PDCS98), p.190–197, ISCA, Chicago, 1998
2. Barth, T., Flender, G., Freisleben, B., Thilo, F. Load Distribution in a CORBA Environment, in: Proc. of Int'l Symposium on Distributed Object and Application 99, p. 158–166, IEEE Press, Edinburgh 1999
3. Barth, T., Grauer, M., Freisleben, B., Thilo, F. Distributed Solution of Simulation-Based Optimization Problems on Workstation Networks. Proc. 2^{nd} Int. Conf. on Parallel Computing Systems, pp. 152–159, Ensenada, Mexico, 1999
4. Boden, H., Gehne, R., Grauer, M., Parallel Nonlinear Optimization on a Multiprocessor System with Distributed Memory, in: Grauer, M., Pressmar, D. (eds.), Parallel Computing and Mathematical Optimization, Springer, 1991, p.65–78.
5. Grauer, M., Barth, T., Cluster Computing for treating MDO–Problems by OpTiX, to appear in: Mistree, F., Belegundu, A. (eds.), Proc. Conference on Optimization in Industry II, Banff, Canada, June 1999
6. Gebauer, C., Load Balancer $\mathcal{LB}$ – a CORBA Component for Load Balancing, Diploma Thesis, University of Frankfurt, 1997
7. Livny, M., Raman, R., High-Throughput Resource Management, in: The GRID: Blueprint for a New Computing Infrastructure, Foster, I., Kesselman, C. (eds.), pp. 311–337, Morgan Kaufmann, 1998
8. Maffeis, S., Piranha: A CORBA Tool for High Availability, IEEE Computer, Vol. 30, No.4, p. 59–66, April 1997
9. Modzelewski, B., Cyganski, D., Underwood, M., Interactive–Group Object–Replication Fault Tolerance for CORBA, 3^{rd} Conf. on Object–Oriented Techniques and Systems, Portland, Oregon, June 1997, pp. 241–244
10. Fault tolerant CORBA, Object Management Group TC Document Orbos/99-12-08, December 1999
11. omniORB – a Free Lightweight High–Performance CORBA 2 Compliant ORB, (http://www.uk.research.att.com/omniORB/omniORB.html), AT&T Laboratories Cambridge, 1998
12. Rackl, G., Load Distribution for CORBA Environments, Diploma Thesis, (http://wwwbode.informatik.tu-muenchen.de/~rackl/DA/da.html), University of Munich, 1997
13. Schiemann, B., Borrmann, L., A new Approach for Load Balancing in High–Performance Decision Support System, Future Generation Computer Systems, Vol: 12, Issue: 5, April 1997, pp. 345-355
14. Schittkowski, K., Nonlinear Programming Codes, Springer, 1980