Implementation Techniques and an Object Group Service for CORBA-Based Applications in the Field of Parallel Processing

3-99

Markus Aleksy, Axel Korthaus

Lehrstuhl für Wirtschaftsinformatik III
Universität Mannheim, Schloß
D-68131 Mannheim, Germany

Tel.: +49 621 181 1642
email: {aleksy|korthaus}@wifo3.uni-mannheim.de

Implementation Techniques and an Object Group Service for CORBA-Based Applications in the Field of Parallel Processing

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

Abstract

At present, several middleware architectures are known for the development of distributed or parallel software applications. While the field of distributed client-server applications is predominated by standards such as the Common Object Request Broker Architecture (CORBA) [13], Remote Method Invocation (RMI) [20] and the Distributed Component Object Model (DCOM) [11], the domain of distributed parallel programming primarily makes use of tools such as the Parallel Virtual Machine (PVM) [6] and the Message Passing Interface (MPI) [12].

In this paper, we examine the suitability of CORBA-based solutions for meeting application requirements in the field of parallel programming. We outline concepts defined within CORBA which are helpful for the development of parallel applications, and we describe which programming techniques are at hand for this purpose. Subsequently, we present an Object Group Service which facilitates the development of CORBA-based, distributed and parallel software applications. Moreover, we introduce some basic ideas of how the Unified Modeling Language (UML) can be used for modeling parallel applications.

1. Introduction

In the field of parallel programming, message passing libraries such as PVM and MPI play a prevailing role. When the Java programming language emerged and became more popular, these solutions were ported to that language and new techniques were developed. There are, for example, products such as JPVM [9], jPVM (formerly JavaPVM) [10], JavaMPI [8], HPJava [7], or DOGMA [2] which aim at enabling parallel and distributed programming with Java. While jPVM accesses the "original" PVM implementation via the Java Native Interface (JNI) [19], JPVM is a purely Java-based solution developed from scratch. A detailed description of JPVM and an assessment of its performance can be found in [5] and [21].

In this paper, we focus on CORBA as a potential middleware solution for distributed parallel programming and we present ways of modeling and implementing CORBA-based parallel applications.

The CORBA standard has been widespread in the area of object-oriented and distributed systems. Not only does it support independence of the computer architectures and programming languages to be used, it allows users a vendor-independent choice of ORB products as well. This last option has been made possible in CORBA 2.0 through the establishment of a unique object reference, the Interoperable Object Reference (IOR), and through a standardized transmission protocol, the Internet-Inter-ORB-Protocol (IIOP). In the course of our research work on CORBA, we analyzed the interoperability of ORB products from different vendors [15] and examined portability and exchangeability of the stubs and skeletons produced by the corresponding Interface Definition Language (IDL) compilers [1]. The results of these examinations have shown that collaborations between different C++ and Java ORB products work well and, as far as Java is concerned, the files generated by the respective IDL compilers can be interchanged in many cases. It came out that the main disadvantage of CORBA is the sheer size and complexity of the core specification [13] and of the numerous Common Object Service Specifications (COSS), which extend the basic functionality of CORBA.

The basic communication model propagated by CORBA [13] is synchronous and blocking. Synchronous communication means that the client invokes a server operation and has to pause its own processing until the server has processed the call and finally acknowledged the termination of operation. An acknowledgement even occurs if the return value of the operation being called is of type `void`. During the

processing of the request on the server, the client blocks, i.e., no further activities can take place on the client. In case of communication errors or server failures, the blocking might last for an undesirably long period of time, since no acknowledgement will arrive from the server and the client will not be able to resume its operation before a certain timeout period has passed which has been specified by the developers or users. Even if no such errors occur, blocking of the client can be very inadequate, e.g., in situations where operation calls lead to complex and time consuming computations. In that case, valuable processing time will unnecessarily be lost for the client if it does not depend on the immediate return of the computed result, but could perform computations and evaluations on its own in the meantime.

If CORBA is to be used as the middleware for a distributed, parallel application, this basic communication model seems to be insufficient in most cases. For example, there might be a "master" process residing on one computer in the network which dispatches tasks that are to be executed in parallel by several "worker" processes living on different computers in the network. In that case, a basic requirement is that it must be possible to trigger the workers via CORBA in an asynchronous way in order to achieve parallelism and enable the master to continue its work.

Therefore, we have to analyze how asynchronous operation invocations can be performed and how client blocking can be avoided in CORBA. The following sections present possibilities of how to achieve this goal. The description is not limited to the means provided by the CORBA standard itself, but also includes techniques for solving this kind of problems in Java.

2. CORBA Features for Asynchronous Operation Invocation

The CORBA standard defines three possible ways of extending the basic synchronous communication model by asynchronous and/or non-blocking calls (cf. [14]), namely:

- the Interface Definition Language (IDL) keyword **`oneway`**,
- the use of the Dynamic Invocation Interface (DII), and
- the use of the Event Service.

The first option is the use of the IDL keyword **`oneway`**. With the help of this keyword it can be specified that a server operation is only receiving information from the client, but does not return any information back. For **`oneway`**-operations, only "**`in`**"- parameters are allowed and the type of the operation result has to be **`void`**. This solution works according to the "fire and forget" principle, i.e., after having sent its request, the client does not block, nor does it receive any kind of notification, whether the request has been processed successfully or not. Should the request have not reached the server, e.g., due to some communication error, the client does not get any feedback about that situation. In our experimental setup, we have tested **`oneway`**-invocations with OmniBroker, OAK and JavaORB on Linux and Solaris. On Linux, these invocations are not performed properly by OmniBroker and OAK, even if client and server are running on the same computer. The request does not show any effect at all; there is even no error message output. Only JavaORB behaves as expected and processes the operation invocations duly. On Solaris, none of the ORBs mentioned above show these problems. Whether the reasons for the undesired behavior of OmniBroker and OAK on Linux have to be contributed to faulty Java Virtual Machines or to errors in the ORBs themselves could not be found out.

The second way of achieving our goal is by employing the Dynamic Invocation Interface. In that case, requests have to be made using a special request operation called **`send_deferred()`** and the results can be obtained by calling an operation named **`get_response()`** at any time anywhere in the program code. Although the invocation is asynchronous, it nevertheless might block the client in case the server has not finished the processing of the request at that time. If, for any reason, the client must not halt when its flow of control reaches that point, the developer has to fall back on the features of the programming language (s)he uses. A different alternative can be a "dynamic" **`oneway`**-invocation, executed with the help of operation **`send_oneway()`**. Since this approach does not provide any feedback, it is superfluous to call **`get_response()`**, and consequently, no blocking of the client can occur.

The Event Service represents the last of the three possibilities of achieving asynchronous and/or non-blocking communication that are available today. Although it does not belong to the CORBA core, it has been added to the OMG standard as a part of the COSS. The Event Service applies the "publish/subscribe"-pattern. Users of the Event Service are subdivided into suppliers and consumers. The so-

called Event Channel represents the central element of this service, on which the following communication models are supported (cf. Fig. 1):

- pure Pull-Model,
- pure Push-Model,
- hybrid Pull/Push-Model,
- hybrid Push/Pull-Model.

In the Pull-Model, the Pull-Consumer has to detect events actively, whereas a Push-Consumer behaving according to the Push-Model is passive, waiting to be informed automatically on the occurrence of an event. The event suppliers show corresponding behavior in the two models, although with reversed roles, i.e., the Pull-Supplier acts passively, while the Push-Supplier actively propagates events to the Event Channel.

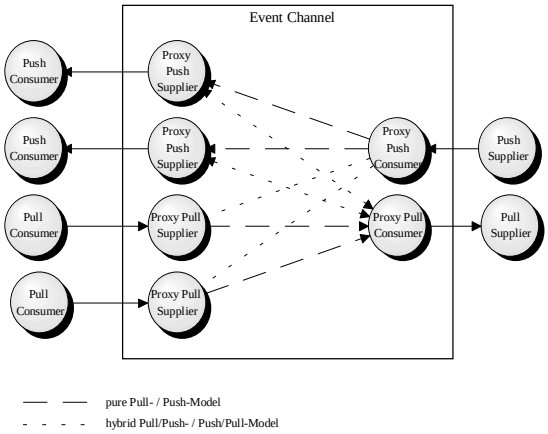

Fig. 1: Different Communication Models Supported by the Event Service

As we have already illustrated in our interoperability analysis [15], an Event Service from a specific vendor can be used together with servers and clients that rely on an ORB from a different vendor without any problems. This can be another reason for using a standardized service instead of developing an individual solution.

However, the Event Service also shows several weaknesses which can play an important role in the context of application development:

- there is no guarantee that an event is actually delivered to an event consumer,
- there is no support of filters, i.e., each event consumer might possibly get a large number of events in which it might not be interested at all,
- there is no additional information such as the current number of event consumers etc.

In order to improve these drawbacks, the OMG has specified a so-called Notification Service, which we are not going to describe here. Further techniques of synchronous and asynchronous communication in CORBA, concerning "CORBA messaging", can be found in [17] and [18].

3. Techniques for Parallel Programming with CORBA

CORBA is a technology for client/server applications. Typical CORBA-based applications consist of many clients communicating with one or more servers. As opposed to this, architectures of parallel distributed applications are often constituted by a single client (called "master") accessing several servers (called "workers") which execute a computation in parallel. In this section, we present several ways of designing CORBA-based parallel applications.

Our first example (**`Example1.idl`**) contains two operations: **`compute()`** and **`get_result()`**. The first operation is used for transferring data needed for the computation from master to workers and to start the computation. In order to avoid blocking of the master, this operation has to be **`oneway`**. With the help of operation **`get_result()`** the master can request the result of the computation afterwards. Since it delivers a return value, this operation must not be **`oneway`** and, thus, blocks the master in case the worker has not finished its computation up to the moment of the invocation of **`get_result()`**.

```
// Example1.idl
interface Worker
{
  typedef sequence<long> l_array;

  oneway void compute(
    in l_array data);
  long get_result();
};
```

A first improvement on this situation could be the introduction of an additional operation **`ready()`** which could be called by the master to find out whether the worker's computation has completed in the meantime. The following IDL-interface (**`Example2.idl`**) illustrates this approach:

```
// Example2.idl
interface Worker
{
  typedef sequence<long> l_array;

  oneway void compute(
    in l_array data);
  boolean ready();
  long get_result();
};
```

Comparable to the approach presented before is the solution that is described in **Example3.idl** below. In this approach, we have dispensed with the additional operation **ready()** and have defined an exception **NotReady** instead, which can be raised by operation **get_result()**.

```
// Example3.idl
interface Worker
{
  typedef sequence<long> l_array;

  exception NotReady { };

  oneway void compute(
    in l_array data);
  long get_result()
    raises(NotReady);
};
```

Although the appearance of the last two approaches is slightly different, they both show the same main disadvantage. In both cases, additional network traffic is produced depending on the frequency of invoking operation **ready()** in **Example2** and of operation **get_result()** throwing exception **NotReady** in **Example3**, respectively.

The main problem is to determine the point of time when each worker has completed its computation. Since each single worker usually only knows his own status, the IDL interface should be modeled accordingly to take into account the specific responsibilities of master and workers. Specifically, two interfaces should be defined: the **Master** interface and the **Worker** interface. The **Worker** interface only contains operation **compute()** which requires the data needed to perform its computation and a CORBA reference to the CORBA object representing the master as invocation arguments. The newly defined **Master** interface contains an operation called **set_result()**. Now, a worker object is capable of notifying the calling master object of the completion of its computation and can deliver the respective result the moment it is available. Thus, superfluous requests for the status of the computation can be omitted and, therefore, the network traffic connected with these requests is avoided. This kind of procedure, i.e., the invocation of a master object's operation by a worker object, is called "callback". The following IDL interface (**Example4.idl**) is to illustrate the callback technique:

```
// Example4.idl
interface Master
{
  void set_result(in long result);
};

interface Worker
{
  typedef sequence<long> l_array;

  oneway void compute(
    in l_array data, in Master ior);
};
```

4. Avoiding Blocking on the Programming Language Level

Taking into account that asynchronous operation calls are connected with certain restrictions, synchronous calls are in many cases to be preferred. As mentioned before, the main disadvantage of synchronous communication techniques results from blockings the master is afflicted with. Fortunately, this problem can be solved on the programming language level, provided a language is used which is suitable for that purpose. For example, if C, C++ or Java are used, then blocking can be avoided by aptly employing multithreading techniques. However, in the case of C and C++ this might lead to restrictions with regard to portability. Furthermore, regardless of the programming language used, the development effort increases and the source code becomes less understandable.

The concept presented here is based on the use of multithreading in Java. Class **LocalWork** represents the prototype of a class which takes over computational tasks in the period of time during which the master has to wait for the workers. In contrast to class **LocalWork**, class **Wait** is responsible for fetching the result of the computation performed by the worker. The invocation of the worker's operation and the request for the result is based on CORBA's Dynamic Invocation Interface (DII). For the sake of simplicity, we have dispensed with parameters and a return value for the **compute()** operation of class

LocalWork. The following code snippet elucidates this technique:

```
class LocalWork {
  void compute() {
    try {
      // do some work
    }
    catch(InterruptedException ex) {
    }
  }
}

class Wait extends Thread {
  private org.omg.CORBA.Request req;
  private boolean ready = false;

  Wait(org.omg.CORBA.Request req) {
    this.req = req;
  }

  public void run() {
    // ready?
    req.get_response();
    ready = true;
  }

  boolean isReady() {
    return ready;
  }
}

class DontBlock {
  private org.omg.CORBA.Request req;
  private org.omg.CORBA.Object obj;
  private static boolean
    returned = false;

  DontBlock(
    org.omg.CORBA.Object obj) {
    this.obj = obj;
  }

  void callMethod() {
    // remote call
    req = obj._request(
      "method_to_call");
    req.send_deferred();

    LocalWork wo = new LocalWork();
    Wait wa = new Wait(req);
    wa.start();

    // work while waiting
    while(!returned) {
      wo.compute();
      returned = wa.isReady();
    }
    returned = false;
  }
}
```

5. Introduction of an Object Group Service (OGS) for Parallel Processing

Due to the currently insufficient specification of the CORBA standard with regard to group communication, we have developed our own Object Group Service (OGS) in order to support the design of applications using parallel processing already on the IDL level.

Our primary goals in the development of the OGS were:

- Simplicity with respect to the handling of the OGS, supported by a restriction of the scope of the OGS to core functionality, and
- Generality of functionality by abstraction from specific data structures in order to make the service suitable for a great variety of tasks.

The general idea of a CORBA-based Object Group Service traces back to the work of Felber ([3], [4]) who has suggested this approach for use in replication scenarios. As opposed to this application purpose, our design and the resulting implementation aim at supporting the parallel processing of CORBA operation calls, i.e., a request issued by a master is propagated to any number of workers in parallel, which process the data transferred independently of each other.

An OGS client has to implement interface **Master** which only contains one single operation, called **receive()**, which gets information issued by a worker. The worker, on the other hand, has to implement interface **Worker** including operation **send()** which not only gets information about the operation to be executed as an argument, but also the Interoperable Object Reference (IOR) of the master. The latter is compulsory in order to be able to make use of the callback technique. As a first step, the master transmits the message to be performed and its own IOR to a certain group of workers, using operation **send()**. The second step comprises the dispatch of the message from the group to the group members, the workers. The last step is represented by the workers calling the master's operation **receive()** in order to inform the master of the result of the computation.

The responsibility of the IDL interface **Group** is to propagate a call issued by the master to all members of the group. It contains functionality for attachment and detachment of workers to/from a group. If a worker tries to register itself although it has already been registered before, an **AlreadyRegistered** exception will be raised. Similarly, an attempt to detach a worker which is not registered leads to a **NotRegistered** exception.

To be able to administer several groups of workers, we have defined a **GroupManager** interface. It provides operations for creation (**create()**), listing (**list()**), determination (**resolve()**), and deletion (**destroy()**) of groups. Operation **create()** will raise an **AlreadyExists** exception in case a group with the name specified is already created, operation **resolve()** will throw a **NotFound** exception in case the group does not exist, and, in the same case, the invocation of operation **destroy()** will lead to a **NotAvailable** exception.

The architectural design of the Object Group Service can be understood by looking at the following IDL interface:

```
module GroupService
{
  exception AlreadyRegistered { };
  exception NotRegistered { };
  exception AlreadyExists { };
  exception NotAvailable { };
  exception NotFound { };

  interface Master
  {
    oneway void receive(
      in any result);
  };

  interface Worker
  {
    oneway void send(in any message,
      in Master ior);
  };

  interface Group : Worker
  {
    void registerWorker(
      in Worker ior)
      raises(AlreadyRegistered);
    void unregisterWorker(
      in Worker ior)
      raises(NotRegistered);
  };

  interface GroupManager
  {
    typedef sequence<string>
      Grouplist;

    Group create(
      in string groupname)
      raises(AlreadyExists);
    Grouplist list();
    Group resolve(
      in string groupname)
      raises(NotFound);
    void destroy(
      in string groupname)
      raises(NotAvailable);
  };
};
```

We have chosen the CORBA type **any** as the data type of the **message** parameter of operation **send()** and of the **result** parameter of operation **receive()**. The advantage of this decision is, that arbitrary data types and structures can be transmitted. On the other hand, a disadvantage results from the fact, that marshalling and unmarshalling of type **any** takes more time than of simple data types such as **long**.

Fig. 2 illustrates the structure and the functionality of the OGS in a graphical way.

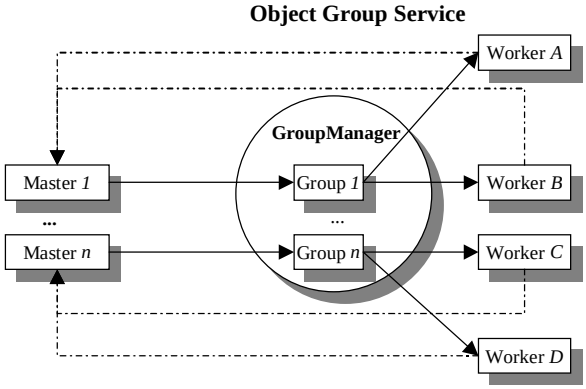

Fig. 2 : Structure and Functionality of the OGS

Let us have a look at an example scenario of the use of the OGS for parallel processing purposes. The UML sequence diagram in Figure 3 shows the dynamic flow of messages in the scenario described below.

In the example, **worker1** creates two groups **g1** and **g2** and registers itself on these groups. With the help

of operation **`list()`**, **`worker2`** finds out which groups exist and registers on them. Last, the master requests a list of the existing groups, sends a message to groups **`g1`** and **`g2`** and finally receives the result of its call via a callback (operation **`receive()`** called by the worker).

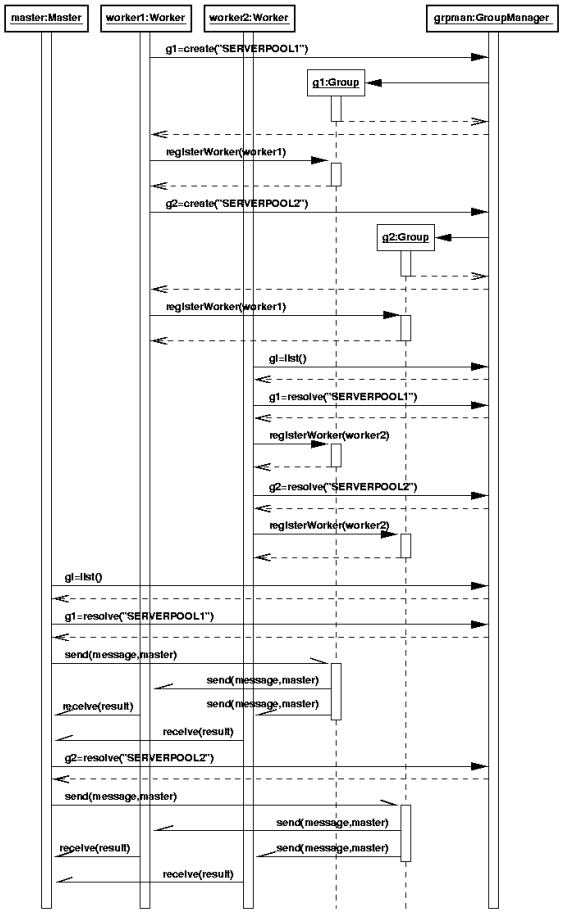

Fig. 3: An Example Scenario of the OGS at Work

The UML sequence diagram shown in Figure 3 gives an idea of how synchronous and asynchronous communication needed for parallel processing can be modeled with UML. Objects which are represented by rectangles with thick frames in the diagram are active objects, i.e., they each have their own thread of control. In the example, these are the **`master`**, **`worker1`**, **`worker2`**, and **`grpman`** objects. **`g1`** and **`g2`** are passive objects which belong to the same process as the group manager, so the frames of their symbols are not thick. To show synchronous operation calls within this environment of concurrently acting objects explicitly, we have used message arrows with filled solid arrowheads. Additionally, dashed return arrows with stick arrowheads show the return from the invocation, after which the processing of the caller can continue. As opposed to those synchronous calls, invocations of operations which can be called asynchronously (indicated by the **`oneway`** keyword in the corresponding IDL interface) are shown by arrows with a half stick arrowhead. In our example, the communication between master and workers, i.e., the dispatching of a message via a group to several workers using operation **`send()`** and the returning of the result using operation **`receive()`**, is done asynchronously. Further aspects concerning modeling concurrent systems with UML can be found in [16].

6. An Example of Using the OGS

Having illustrated the basic dynamic communication behavior of a parallel system using the OGS with the help of a UML sequence diagram, we now show some interesting code snippets of typical master and worker implementations.

Normally, an application that uses the OGS provides an IDL interface that has to be derived from the OGS interface. The **`typedef`** statement that can be seen in the IDL interface code below is very important. It is needed in order to have special helper classes generated (in this case class **`l_arrayHelper`**) by the IDL compiler which provide operations for insertion and extraction of the self-defined data type into and from type **`Any`**. The following IDL interface illustrates this approach:

```
// Demo of an OGS-based application
#include "group.idl"

typedef sequence<long> l_array;

interface DemoMaster :
  GroupService::Master
{
  // additional operations
};

interface DemoWorker :
  GroupService::Worker
{
  // additional operations
};
```

Our first example code snippet shows how the operation **`send()`** which is provided by the worker objects, can be implemented. The first step in our example is to extract a vector of integers from data type

Any, using the generated helper class' operation **extract()**. Afterwards, the computation is performed (which, in our example, simply computes the arithmetic average of the integers in the vector) and the result of type **double** is inserted into type **Any** again. Since the general process of extraction, computation, insertion, and return of the result remains the same independent of the actual data types or data structures used, the example provides a good overview of the way operation **send()** can be implemented.

```
public void send(Any message,
  Master master)
{
  // extract data
  l_arrayHelper data =
    new l_arrayHelper();
  int[] vec = data.extract(message);

  // compute
  double avg = 0;
  for(int i=0; i<vec.length; i++)
    avg += vec[i];
  avg /= vec.length;
  System.out.println("Average = "
    + avg);

  // return result
  org.omg.CORBA.Any result =
    orb.create_any();
  result.insert_double(avg);
    master.receive(result);
  }
}
```

The second example illustrates how a client can send messages to several groups. First, a reference to one of the existing groups has to be obtained from the **GroupManager**. Subsequently, the data is inserted into a variable of type **Any**, then it is transmitted to the group by a **send()** message, and finally it is automatically and transparently dispatched to all the members of the group.

```
// send data to group 1
  Group g1 =
    grpman.resolve("SERVERPOOL1");
  org.omg.CORBA.Any message =
    orb.create_any();
  int[] vec1 = { 1, 2, 3, 4, 5,
                 6, 7, 8, 9 };
  l_arrayHelper data =
    new l_arrayHelper();
  data.insert(message,vec1);
  g1.send(message, master);

  // send data to group 2
  Group g2 =
    grpman.resolve("SERVERPOOL2");
  int[] vec2 = { 10, 20, 30, 40, 50,
                 60, 70, 80, 90 };
  data.insert(message,vec2);
  g2.send(message, master);
```

7. Advantages of the Object Group Service Over the Event Service

Compared with the CORBA Event Service, our Object Group Service not only provides the possibility of broadcast communication, i.e., a call from the master is propagated to all the workers, but also has the following advantages:

- Greater simplicity of use, because the IDL interface definition is designed to be much more understandable,
- Support of multicast communication, i.e., a call from the master can be propagated to a certain number of workers (a group of workers). Using the Event Service, this can only be achieved by starting multiple instances of the server providing the Event Service. Therefore, our solution is less resource consuming, since only one Object Group Service instance has to be active and operation calls are only propagated to those workers responsible for this specific task without the necessity of a filter mechanism that would again be consuming processing time.
- Reduced load for the network and the service itself, since there is no need for an indirect communication channel as in the CORBA Event Service. Due to the callback mechanism used in our solution, the results of the computations are returned to the master directly without involving the OGS.

8. Possible Ways of Implementing the Object Group Service

The OGS can be implemented in several different ways (cf. Figure 4).

The first alternative is to implement the OGS, like all other COSS services, as an independent program that can be run on any computer in the network. A considerable disadvantage of this approach is that the transmission of data from the master to the service and then from the service to the workers is directed over

the network, thus causing a network load nearly twice as high as direct calls to the workers without an intermediate service.

A second alternative could be an implementation in the form of a library that can be called locally and subsequently dispatches the calls to the workers. In that case, the network load would be equal to the load produced if the service would not be used. A major disadvantage would be, however, that the workers would have to be attached to each and every local OGS that needs to use them. Increased programming efforts and higher susceptibility to errors would be the consequence.

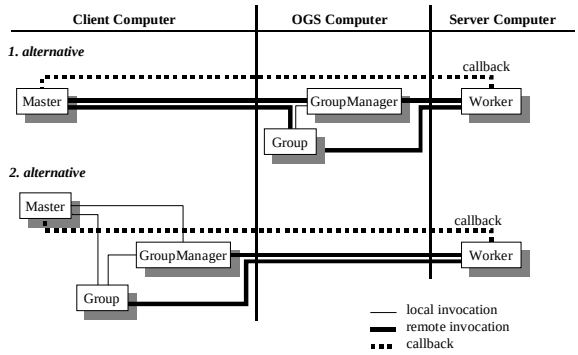

Fig. 4: Implementation Alternatives for the OGS

Figure 5 shows a UML Deployment Diagram corresponding to the first implementation alternative described above. It is an instance level diagram depicting a snapshot of a system implementing the OGS at runtime.

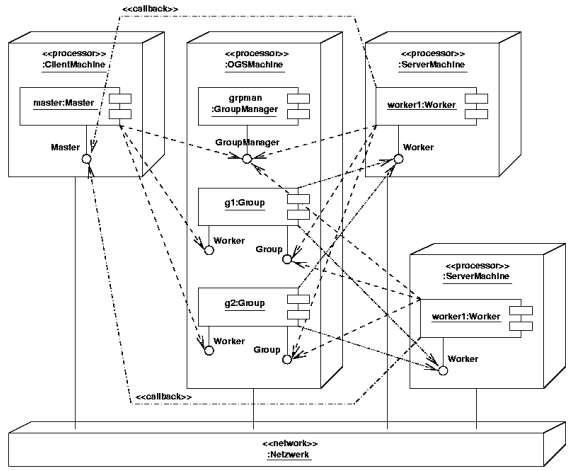

Fig. 5: UML Deployment Diagram of the OGS Scenario

The different computational nodes, e.g., the computer on which the master is running, the network etc., are symbolized by 3D boxes. The CORBA objects implementing master, workers, and the OGS are modeled as UML components with explicit interfaces (shown in "lollipop" notation) which represent their IDL interfaces. The solid lines between the nodes show hardware connections used for the actual flows of information. Dependencies between the components are expressed by dashed lines. The dashed lines show which interfaces are used by which components. For example, the **`master`** component depends on the **`GroupManager`** interface of the **`:GroupManager`** component in order to be able to receive a list of available groups by calling operation **`list()`**.

9. Critical Review of the Approaches Presented

The big advantage of using asynchronous operation calls on the IDL level lies in the independence of the programming language used for the implementation. Thus, asynchronous calls that do not entail blockings can be made from COBOL, Ada, Smalltalk, C++, and Java, likewise. An important drawback is the dependency on the specific ORB product in use. As mentioned before, **`oneway`** calls performed on the basis of OmniBroker and OAK on Linux were not successful.

In the case of Java, solutions on the programming language level are relatively easy and comfortable. The Java language has a built-in support for multithreading, so that only few extensions have to be made and the code remains portable. For small applications or ORBlets, this kind of solution should be preferred, because thus, no callbacks are needed. In the case of ORBlets, callbacks might even be impossible, depending on the deployment structure of the system. For other programming languages than Java or in case several programming languages are being mixed within the same application, solutions on the IDL level are to be preferred.

With its Event Service, the OMG has defined a versatile instrument for enabling asynchronous communication. However, in the context of small applications, its scope of functionality is too comprehensive, so that other solutions have to be considered. A major weakness of the Event Service is its insufficient support for multicast communication.

This drawback is remedied by our Object Group Service. Another advantage over the Event Service is

the low complexity of the applications using the OGS which not only decreases the period of time needed to become familiar with the application code but also leads to clear application structures and ease of maintenance.

References

[1] Aleksy, M., Schader, M., Tapper C. (1999): "Interoperability and Interchangeability of Middleware Components in a Three-Tier CORBA-Environment – State of the Art", in: Proceedings Third International Enterprise Distributed Computing Conference EDOC'99, 27-30 Sept. 1999, University of Mannheim, Germany, IEEE, Piscataway, New Jersey, pp. 204-213

[2] DOGMA (1999): Distributed Object Group Metacomputing Architecture (DOGMA) Webpage, http://ccc.cs.byu.edu/DOGMA/System.html

[3] Felber, P., Garbinato, B., Guerraoui R. (1996): "The design of a CORBA group communication service"; in: Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, pp. 150-159, ftp://ftp-lse.epfl.ch/pub/felber/papers/SRDS-96.ps

[4] Felber, P., Guerraoui R., Schiper A. (1998): "The implementation of a CORBA group communication service"; in: Theory and Practice of Object Systems, vol. 4, no. 2, pp. 93-105

[5] Ferrari A. J. (1998): "JPVM: Network Parallel Computing in Java" http://www.cs.virginia.edu/~ajf2j/jpvm.html

[6] Geist, A., et al. (1994): PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, Massachussetts, http://www.netlib.org/pvm3/book/pvm-book.html

[7] HPJava (1999): The HPJava Project Webpage, http://www.npac.syr.edu/projects/pcrc/HPJava/

[8] JavaMPI (1999): JavaMPI Webpage, http://perun.hscs.vmin.ac.uk/JavaMPI/

[9] JPVM (1999): JPVM—The Java Parallel Virtual Machine Webpage, http://www.cs.virginia.edu/~ajf2j/jpvm.html

[10] jPVM (1999): jPVM Webpage (previously JavaPVM), http:// www.isye.gatech.edu/chmsr/jPVM/

[11] Microsoft Inc. (1999): "Distributed Component Object Model (DCOM)"; General Microsoft web site containing links to information about the DCOM Technology, http://www.microsoft.com/com/dcom.asp

[12] MPI (1999): General web site containing official standard documents about the Message Passing Interface, http//www.mpi-forum.org/

[13] OMG (1998): "CORBA/IIOP 2.2 Specification"; OMG Technical Document Number 98-07-01, http://www.omg.org/corba/corbaiiop.html

[14] OMG (1997): "Event Service Specification"; OMG Technical Document Number 97-12-11, ftp://www.omg.org/pubs/docs/format/97-12-11.pdf

[15] Schader M., Aleksy M., Tapper C. (1998): "Interoperabilität verschiedener Object Request Broker nach CORBA2.0-Standard"; in: OBJEKTspektrum, 3/98, pp. 72-77, http://www.wifo.uni-mannheim.de/ IIOP

[16] Schader M., Korthaus A. (1998): "Modeling Java Threads in UML"; in: Schader M., Korthaus, A. (eds.): The Unified Modeling Language – Technical Aspects and Applications, Physica, Heidelberg, pp. 122-143

[17] Schmidt D. C., Vinoski S. (1998): "An Introduction to CORBA Messaging"; in C++ Report, SIGS, vol. 10, no. 10

[18] Schmidt D. C., Vinoski S. (1999): "Programming Asynchronous Method Invocations with CORBA Messaging"; in C++ Report, SIGS, vol. 11, no. 2

[19] Sun Microsystems Inc. (1997): "Java Native Interface Specification"; JDK 1.1, May 16, 1997, http://java.sun.com/products/jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html

[20] Sun Microsystems Inc. (1998): "Java Remote Method Invocation Specification"; Revision 1.50, JDK 1.2, Oct. 1998, http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[21] Yalamanchilli N., Cohen W. (1998): "Communication Performance of Java based Parallel Virtual Machines" http://www.cs.virginia.edu/~ajf2j/jpvm.html