A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming

2-00

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

A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming

Markus Aleksy
Wirtschaftsinformatik III
University of Mannheim
Schloß (L 5,6)
68131 Mannheim, Germany
+49 621 181 1488
aleksy@wifo3.uni-mannheim.de

Axel Korthaus
Wirtschaftsinformatik III
University of Mannheim
Schloß (L 5,6)
68131 Mannheim, Germany
+49 621 181 1641
korthaus@wifo3.uni-mannheim.de

ABSTRACT
The field of distributed parallel programming is predominated by tools such as the Parallel Virtual Machine (PVM) [4] and the Message Passing Interface (MPI) [10]. On the other hand, mainly standards like the Common Object Request Broker Architecture (CORBA) [11], Remote Method Invocation (RMI) [18], and the Distributed Component Object Model (DCOM) [9] are used for distributed computing.

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. Subsequently, we present our design of an Object Group Service and a Join Service which facilitate the development of CORBA-based distributed and parallel software applications by transparently encapsulating typical forking and joining mechanisms often needed in that context.

Keywords
CORBA, parallel processing, group communication

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 [7], jPVM (formerly JavaPVM) [8], JavaMPI [6], HPJava [5], 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) [17], 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 [3] and [19].

In this paper, we focus on CORBA as a potential middleware solution for distributed parallel programming and we present our approach to the transparent distribution and joining of data in accordance with different strategies for these tasks.

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 [13] 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.

The basic communication model propagated by CORBA 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 the 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. [12]), namely:

- the Interface Definition Language (IDL) keyword **`oneway`**,
- the Dynamic Invocation Interface (DII), and
- 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`**. Calls to operations which have been specified as being **`oneway`** are executed according to "best-effort" semantics, i.e., there is no guarantee that these messages will be actually delivered. It is only guaranteed that a call will be delivered at most once. There are no further details with respect to the semantics of **`oneway`** calls in the CORBA standard, although the standard obviously seems to imply that these calls are asynchronous and non-blocking. As a matter of fact, all the ORB products we have analyzed have the common understanding that **`oneway`** calls should be implemented to show asynchronous and non-blocking behavior, 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.

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 used. 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 Common Object Services Specification (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:

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

As we have already illustrated in our interoperability analysis [13] 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. But since the Event Service shows several weaknesses, the OMG has specified an enhanced solution, the 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 [15] and [16].

3 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. 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.

4 THE COMPONENTS OF OUR SOLUTION

The solution we suggest consists of:

- an Object Group Service (OGS),
- a Join Service,
- one or more masters, and
- one or more workers.

In this scenario, it is the responsibility of the OGS to pass a request issued by a master to the members of a group of workers. The Join Service is in charge of collecting the workers' results, combining them and sending a single answer back to the master.

Fig. 1 illustrates the structure and the functionality of the OGS and Join Service approach in a graphical way.

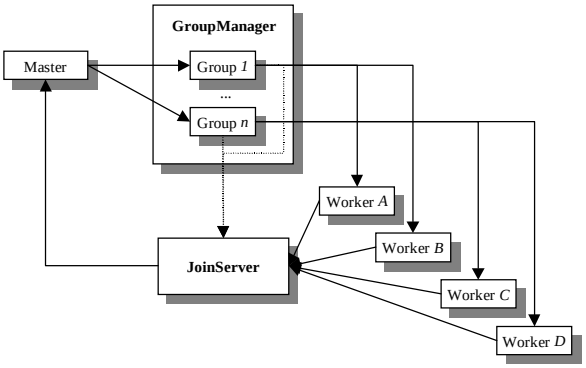

Fig. 1 : Structure and Functionality of the OGS- and Join Service Approach

In the following section, we will explain the components of our solution in detail.

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,
- generality of functionality by abstraction from specific data structures in order to make the service suitable for a great variety of tasks, and
- support of several different data dispatching strategies in order to allow different computing algorithms and to decrease network traffic (compared to sending the complete data to each worker).

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 an interface called **`Master`**, containing only one single operation, called **`receive()`**, which gets information issued by the Join Server. A 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 is also provided with the Interoperable Object Reference (IOR) of the Join Server as well as some information about the call itself. The IOR is compulsory in order to be able to send the result back to the master via a callback. As a first step, the master asynchronously transmits the message to be performed, the dispatching strategy of choice together with additional information, its own IOR, and a particular joining strategy to a particular group of workers by calling **`oneway`**-operation **`send()`**. The second step comprises informing the Join Server of the prospect of getting results from the workers, and dispatching the message to the group members, the workers. The last step is represented by the workers calling the Join Service's operation **`receive()`**, where the results of the computations are combined to a single end result that can be sent back to the master.

The responsibility of operation **`send()`** specified in IDL interface **`Group`** is to propagate a call issued by the master to all members of the group. The interface describes functionality for attachment and detachment of workers to and from a group, respectively. 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. With the help of operation **`get_size()`** the number of workers belonging to a specific group can be determined. The result of this operation can be helpful when the concrete dispatching strategy has to be decided upon.

By calling operation **`merge()`**, the Join Service is informed of the event of sending a master's request to its members, i.e. the workers registered with the group. Furthermore, this message contains additional information allowing a quicker identification of the incoming worker messages with respect to the different groups and the multicasts involved.

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 **`AlreadyEx-`**

ists 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.

A **DataStructure** contains the data that has to be distributed as well as information about the dispatching strategy that has been chosen (see column on the right). A **JoinStrategyInfo** contains information relevant for the recombination of data.

Both the **result** parameter of operation **receive()** and the **data** element inside **DataStructure** which is used as the **message** parameter of operation **send()** are of CORBA type **any**. The reason for this design decision is, that it is thus possible to transmit arbitrary data types and structures through the OGS. The price for this flexibility is a certain loss in performance, because marshalling and unmarshalling of type **any** takes more time than that of simple data types.

As we have already pointed out in section 1, it is desirable to minimize the blocking of the master. Therefore, by specifying operation **send()** of the **Group** interface as **oneway**, we use one of CORBA's built-in mechanisms for asynchronous communication. During our studies, we tested another approach, based on a synchronous call to the OGS, but applying multithreading in order to minimize the master's blocking. In that approach, operation **send()** simply adds the data arguments to a queue and returns immediately, thus keeping the blocking time of the master very low. For each group of workers, a dispatcher thread is started which reads the queue and dispatches the data to the workers.

However, a performance evaluation of the two solutions came to the result that, because of the reduced communication overhead, the asynchronous solution is about 60 times faster than the synchronous one based on multithreading. In our test environment, we had a master on a Pentium III PC with Linux 2.2.10 perform 100 calls to each version of the OGS, which were located on a SPARCstation 10 with Sun Solaris 2.5, together with the Join Service and five workers (connection via 10 MBit Ethernet). The synchonous calls took 14092 ms, while the asynchronous calls only required 230 ms. In the following examples, we refer to the asynchronous solution only.

6 DATA DISPATCHING STRATEGIES

The design of our Object Group Service allows the distribution of data independently of the data types or structures at hand. The developer only has to pay attention to the requirement that the data has to be organized as a CORBA sequence, i.e., a vector with variable length, in case a different strategy than simply passing the whole data to all workers is to be applied. The data elements of the sequence, on the other hand, can have any simple or complex structure. Simple basic data types are just as permissible as complex data types containing other data types such as simple types, arrays, sequences, self-defined (and possibly layered) structures. At runtime, the OGS dynamically determines the type of the data contained in the CORBA sequences and copies the data into new subsequences of the same data type, created in consideration of the number of workers and the number of data sets to be dispatched. A positive aspect of this approach is the possibility to distribute even sequences of data types/structures that were not known when the OGS was developed. Thus, the OGS is capable of covering a broad field of current and future applications.

Element **dds** in a **DataStructure** serves for specifying the dispatching strategy for the data to be distributed. Furthermore, a **DataStructure** contains information needed for the execution of the chosen strategy. In addition to the element **dds** mentioned before, there are some strategy-specific components. The array **data_size** provides information necessary for the execution of the **DIFFERENT_SIZE** strategy (see below). The array **data_length**, on the other hand, is used if the **SAME_SIZE** strategy applies.

The following snippet from the IDL interface of the Object Group Service illustrates this:

```
enum DataDispatchStrategy
{
  ANYTHING,
  PER_ELEMENT,
  SAME_SIZE,
  DIFFERENT_SIZE
};

typedef sequence<unsigned long>
  sizeSeq;

struct DataStructure
{
  DataDispatchStrategy dds;
  sizeSeq data_size;
  unsigned long data_length;
  any data;
};
```

The OGS supports the following data dispatching strategies:

- **ANYTHING**:
  The OGS sends the complete data sets to all workers.
- **PER_ELEMENT**:
  Here, the first element is sent to the first worker, the second element to the second worker, and so on. When

the last worker has received its data the cycle starts again with the first server (provided that there is still more data to be distributed), i.e., given that there are **n** workers, the first worker gets element **n+1**, the second worker gets element **n+2** and so on, until there are no more elements left.

- **SAME_SIZE**:
  With this strategy, the OGS uses the value of **data_length** as the specification of the number of data elements to be sent to each worker. The quantity of data specified in **data_length** is assigned to each worker, e.g., if **data_length=3**, the first worker gets the first three data sets, the second worker gets the next three data sets, and so on, until no data sets are left. Should the number of data sets to be transmitted exceed the product of the number of workers and **data_length**, the surplus data sets are lost. In the opposite case, some workers might receive no data at all.
- **DIFFERENT_SIZE**:
  If the **DIFFERENT_SIZE** strategy is used, the **data_size** component is evaluated to determine the individual quantity of data sets to be assigned to each specific worker. The sequence **data_size** consists of an array of integers whose values reflect the number of data sets the corresponding worker has to process. This means that **data_size[i]** contains the number of elements to be delivered to worker **i+1**. It is permissible to have the value **0** contained in the sequence, in which case the corresponding worker does not get any data. Should the actual number of data sets to be distributed be bigger than the sum of the integer values contained in **data_length**, the additional data sets are simply neglected. In the opposite case, the specified quantities are being delivered as long as there are data sets left. If the OGS runs out of data, the dispatching is terminated, so that some workers might possibly remain unconsidered.
- Default:
  If the OGS is not able to recognize the data format provided (i.e., if it is not a CORBA sequence), it will use its default dispatching strategy which is equivalent to the **ANYTHING** strategy. Thus, the complete data will be sent to all the workers registered with the service.

Table 1 shows an example consisting of three workers and a simple data sequence: {1, 2, 3, 4, 5}. The rows of the table illustrate how the data will be distributed between the three workers depending on the kind of dispatching strategy in use.

Strategy	Worker 1	Worker 2	Worker 3
PER_ELEMENT	1, 4	2, 5	3
DIFFERENT_SIZE { 2, 1, 7 }	1, 2	3	4, 5
SAME_SIZE { 3 }	1, 2, 3	4, 5	
ANYTHING	1, 2, 3, 4, 5	1, 2, 3, 4, 5	1, 2, 3, 4 5
UNKNOWN	1, 2, 3, 4, 5	1, 2, 3, 4, 5	1, 2, 3, 4, 5

Table 1: Distribution of Data Resulting from Different Dispatching Strategies

7 THE JOIN SERVICE

The Join Service represents the counterpart of the OGS. Its function is to collect and combine the results of the group requests that have been sent to the workers. Subsequently, the end result of the combination can be sent back to the master. To be able to perform this task, the Join Service needs some information from the master and from the OGS. The information originating from the master is about which strategy has been chosen for joining the resulting data. Presently, the following joining strategies are supported:

- **COMPLETE** and
- **PARTIAL**.

The **COMPLETE**-strategy has the effect that the Join Service will wait for the results of all the workers involved. In case of a crash of one or more workers during computation, this means that no result will be reported back to the Join Service by those workers and the service threads waiting for the results will have to wait infinitely.

Using the **PARTIAL**-strategy, this drawback can be avoided. In addition to information about the strategy itself, a timeout value has to be specified for the service. In this case, the Join Service maximally waits for the specified period of time and, subsequently, reports back the results of those workers which have already delivered. If all the workers send their answers before timeout, the Join Service will report the complete end result back to the master immediately, not waiting for the end of the timeout period.

The data about the joining strategy is transferred from the master to the OGS which adds its own information about the group identity, the job identity and the identities of the workers involved and transmits the extended data to the Join Service. This additional information is needed for an accelerated identification of the incoming worker messages. The sequence of messages reaching the Join Service may be arbitrary; because of the identification information the service will be capable of recombining the results accordingly, no matter whether the first worker is also the first one to reply or not.

8 AN OGS SCENARIO MODELED WITH UML

Figure 2 shows a UML Deployment Diagram corresponding to our implementation. It is an instance level diagram

depicting a snapshot of a system implementing the OGS and Join Service approach at runtime.

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, the OGS and the Join Service 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, e.g., a component uses an interface of another component. 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()`**.

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

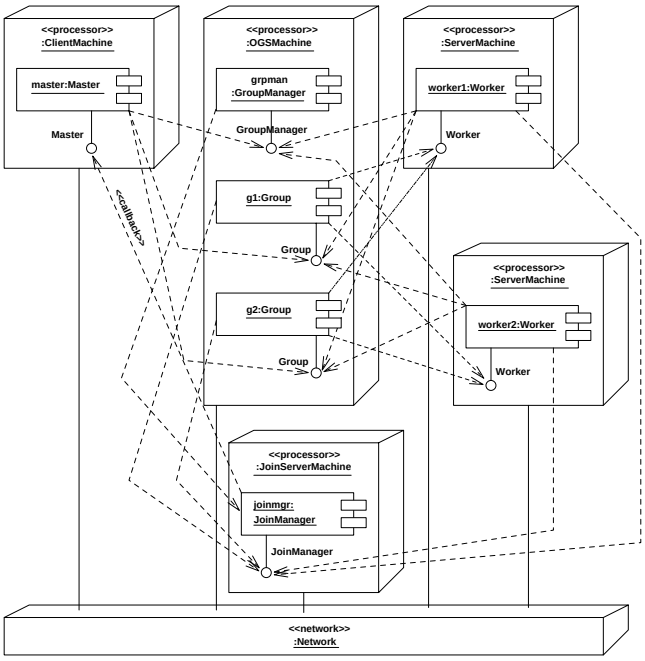

Fig. 2: UML Deployment Diagram of the OGS Scenario

In the example, **`worker1`** creates two groups **`g1`** and **`g2`** and registers itself with these groups. With the help of operation **`list()`**, **`worker2`** finds out which groups exist and registers with them. Last, the master requests a list of the existing groups and sends a message to groups **`g1`** and **`g2`**, from where the data is dispatched to the workers and the information required for recollecting the data is transmitted to the Join Manager.

The UML sequence diagram shown in Fig. 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 [14].

9 AN EXAMPLE OF USING THE OGS WITH THE JOIN SERVICE

Having illustrated the basic dynamic communication behavior of a parallel system using the OGS and the Join Service 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 based on the OGS interface. The **`typedef`** statement 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 SimpleMaster :
  GroupService::Master
{
  // additional operations
};
```

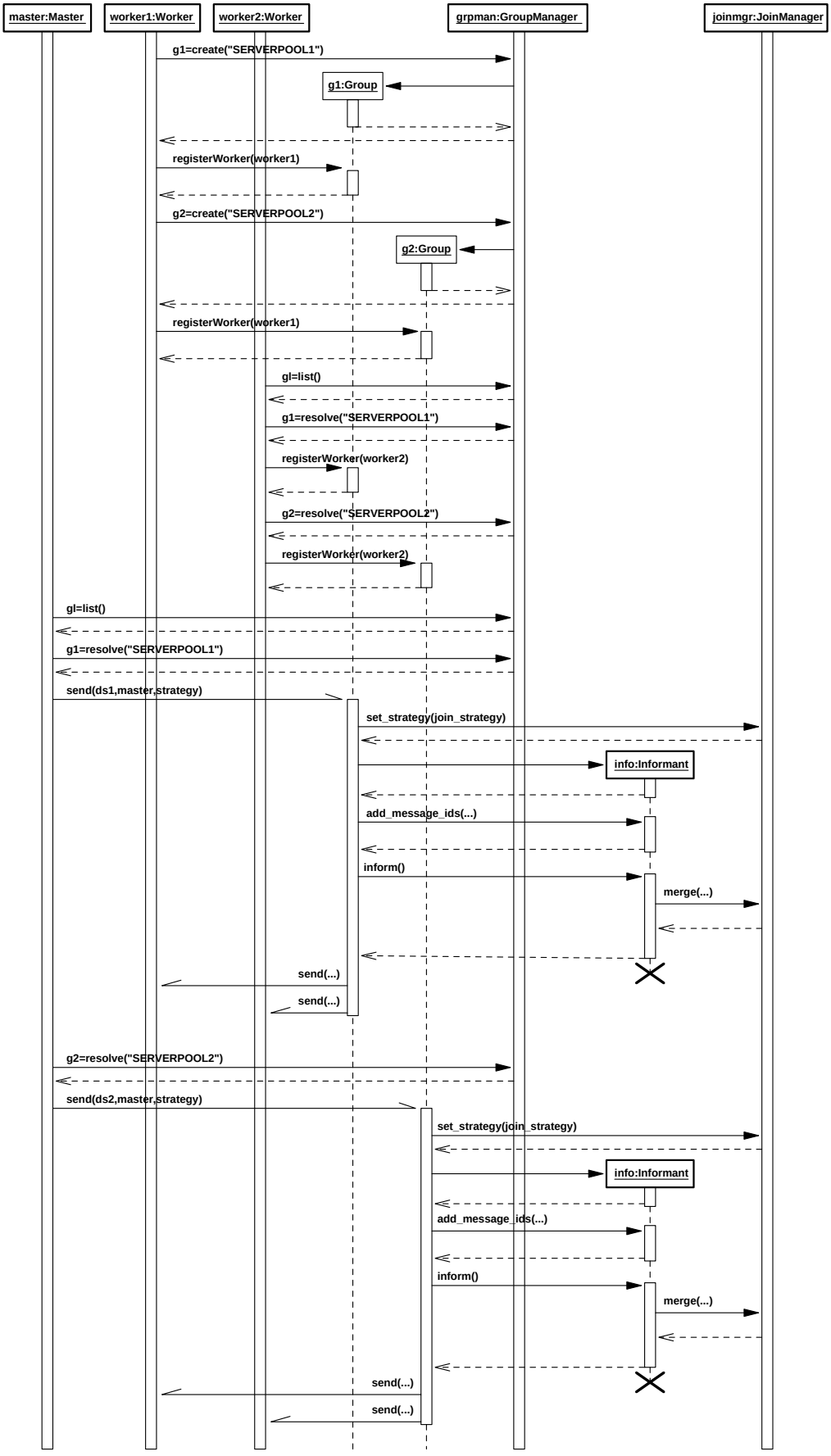

Fig. 3: An Example Scenario of the OGS at Work

```
interface ExtendedWorker :
  GroupService::Worker
{
  any get_status();
  void set_result(in any data);
  // further operations
};
```

Direct communication between workers is not supported by our solution. It can be made possible, if, for example, a special interface is derived from the worker interface providing additional attributes and operations (cf. the IDL listing above) which enable the direct communication between workers. Since this field of duty is not under control of the OGS or the Join Service, the programmer has to take care of the integration of this kind of information exchange and of suitable reactions to possible errors in his application code.

Our first example code snippet below 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 and the result of type **`double`** is inserted into type **`Any`** again. In our example, the computation serves for preparing the determination of the vector norm ($\|u\| = \sqrt{u \cdot u} = \sqrt{u_1^2 + u_2^2 + ... + u_n^2}$) by summing up the squared vector components the respective worker has received. Then, the master only has to sum up the partial results produced by the workers and compute the square root of the total. 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(
  DataStructure message,
  JoinManager ior,
  int group_id,
  int mcast_id,
  int message_id)
{
    // extract data
    l_arrayHelper data =
      new l_arrayHelper();
    int[] vec =
      data.extract(message.data);

    // compute
    double norm = 0;
    for(int i=0; i<vec.length; i++)
      norm += vec[i]*vec[i];

    // return result
    org.omg.CORBA.Any result =
      orb.create_any();
    result.insert_double(norm);

    ior.receive(any, group_id,
      mcast_id, message_id);
  }
}
```

The way messages from the master can be sent to several groups is shown in the second example. Initially, a reference to an existing group has to be obtained from the **`GroupManager`**. Afterwards, the dispatching strategy for the data sets has to be selected. For the first call to operation **`send()`**, the strategy of choice is **`DIFFERENT_SIZE`**. Therefore, an array which holds the quantities of data sets to be delivered to each worker has to be created and initialized. In the example below, the array **`lenSeq`** is filled with the integer values 5 and 4, i.e., the first worker will get five and the second will get four data sets. The second invocation of **`send()`** uses the **`SAME_LENGTH`** strategy which does not require the parameter **`lenSeq`**, so that it could be replaced by a **`null`** reference. Since CORBA does not allow **`null`** references as arguments of operation calls, it is compulsory to fill all the components with non-**`null`** values, independently of whether the specific component will be used or not during execution of the selected strategy.

After choosing the dispatching strategy and the joining strategy, which is the **`COMPLETE`**-strategy for both invocations, the data is inserted into a variable of type **`Any`**, and then it is transmitted to the group via the **`send()`** message. Finally, it is automatically and transparently dispatched to all the members of the group in accordance with the selected strategy.

```
// set joining strategy
JoinStrategyInfo strategy =
  new JoinStrategyInfo(
    JoinStrategy.COMPLETE,0);

// send data to group 1
Group g1 =
  grpman.resolve("SERVERPOOL1");
org.omg.CORBA.Any data =
  orb.create_any();

int[] vec1 = {  2,  3,  5,  7, 11,
               13, 17, 19, 23 };
l_arrayHelper data_helper =
  new l_arrayHelper();
data_helper.insert(data,vec1);
```

```
int[] lenSeq = new int[2];
lenSeq[0]=5;
lenSeq[1]=4;
DataStructure message =
  new DataStructure(
    DataDispatchStrategy.
    DIFFERENT_SIZE,
    lenSeq,0,any);
g1.send(message, master, strategy);

// send data to group 2
Group g2 =
  grpman.resolve("SERVERPOOL2");

int[] vec2 = {  2,  3,  5,  7, 11,
               13, 17, 19, 23 };
data_helper.insert(data,vec2);

message = new DataStructure(
  DataDispatchStrategy.SAME_SIZE,
  lenSeq,5,any);
g2.send(message, master, strategy);
```

10 MULTILEVEL USE OF THE OGS AND THE JOIN SERVICE

The use of the combination of the OGS and the Join Service is not restricted to one computation cycle. Instead, it is possible to apply the services in an iterative way for multilevel computations. In that case, the master sends the data to be dispatched to the OGS (1) which partitions them according to the dispatching strategy and passes them to the workers (2). Their results are collected by the Join Service and the end result is forwarded to the master (4). Thereafter, the results that have been received by the master can be dispatched via the OGS again (5), (6), e.g. to a different group of workers which might perform a totally different kind of processing than the first group of workers. Again, the workers' results (7) are collected and recombined by the Join Service and sent back to the master (see Fig. 4). This process can be iterated as often as needed.

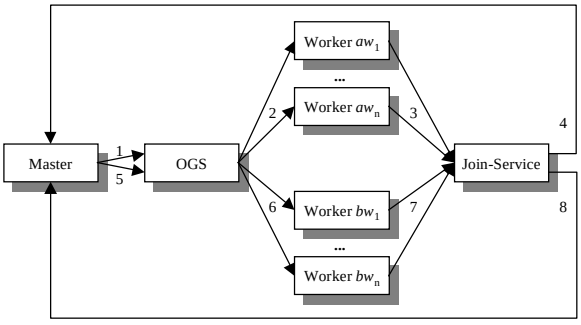

Fig. 4: Multilevel use of the OGS and the Join Service

11 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
- Increased flexibility of data distribution which not only improves the usability of the service but also leads to a significantly reduced network traffic, since each worker only receives its designated data, provided the applied strategy is neither the default one nor the `ANYTHING` strategy. The number of bytes actually transmitted is maximally twice the number of bytes of the data to be dispatched, i.e., it is independent of the number of workers that contribute to the parallel processing. In the case of the Event Service, all registered event consumers get all the data, whether they are interested in the data or not. Thus, the network traffic increases with an increasing number of workers and sums up to `(n+1)*m` bytes with `m` being the number of bytes of data to be transmitted and `n` being the number of event consumers.

12 CRITICAL REVIEW

The solution presented in this paper offers several interesting advantages. The OGS and Join Service approach allows the parallel processing of data independently of the programming language in use. The concurrency of processing on several workers is transparent to the user, i.e., 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 comfortably can benefit from our approach and develop parallel applications even so. Furthermore, it is helpful for those developers who do not have

much or any experience of parallel programming. They do not have to bother with data dispatching and synchronization tasks, because our services externally appear to receive and deliver only simple, single operation calls.

Because of the different data dispatching strategies supported by the OGS, the different joining strategies supported by the Join Service, and the employment of the CORBA data type any as the basic data type for data dispatching, it is possible to use both services for a multitude of applications. Especially the different data dispatching strategies featured allow the reuse of existing computational algorithms with no or only few modifications in most cases.

Another, not yet mentioned field of application for our solution are applets. Provided, that no complicated solutions such as special client policies, signed applets etc. are to be used, applets are generally bound to the restriction that they are only allowed to communicate with the computer from which they have been downloaded. Therefore, a direct access to multiple parallel servers is not possible. With the OGS and the Join Service, applets can play the role of a master, since a master only has to communicate with the OGS and the Join Service. Prerequisite of this architecture is of course, that the OGS and the Join Service run on the same machine as the web server from which the master applet has been loaded.

ACKNOWLEDGEMENTS

We would like to thank the Computing Center of the University of Mannheim, especially Dr. H. Kredel, for their support.

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. Ferrari A. J. (1998): "JPVM: Network Parallel Computing in Java" http://www.cs.virginia.edu/~ajf2j/jpvm.html
4. 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
5. HPJava (1999): The HPJava Project Webpage, http://www.npac.syr.edu/projects/pcrc/HPJava/
6. JavaMPI (1999): JavaMPI Webpage, http://perun.hscs.vmin.ac.uk/JavaMPI/
7. JPVM (1999): JPVM—The Java Parallel Virtual Machine Webpage, http://www.cs.virginia.edu/~ajf2j/jpvm.html
8. jPVM (1999): jPVM Webpage (previously JavaPVM), http:// www.isye.gatech.edu/chmsr/jPVM/
9. 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
10. MPI (1999): General web site containing official standard documents about the Message Passing Interface, http//www.mpi-forum.org/
11. OMG (1998): "CORBA/IIOP 2.2 Specification"; OMG Technical Document Number 98-07-01, http://www.omg.org/corba/corbaiiop.html
12. OMG (1997): "Event Service Specification"; OMG Technical Document Number 97-12-11, ftp://www.omg.org/pubs/docs/format/97-12-11.pdf
13. 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
14. 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
15. Schmidt D. C., Vinoski S. (1998): "An Introduction to CORBA Messaging"; in C++ Report, SIGS, vol. 10, no. 10
16. Schmidt D. C., Vinoski S. (1999): "Programming Asynchronous Method Invocations with CORBA Messaging"; in C++ Report, SIGS, vol. 11, no. 2
17. 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
18. 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/uide/rmi/spec/rmiTOC.doc.html
19. Yalamanchilli N., Cohen W. (1998): "Communication Performance of Java based Parallel Virtual Machines" http://www.cs.virginia.edu/~ajf2j/jpvm.html