Design and Implementation of a CORBA-Based Object Group Service Supporting Different Data Dispatching Strategies

1-00

Markus Aleksy, Axel Korthaus, Martin Schader

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

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

Design and Implementation of a CORBA-Based Object Group Service Supporting Different Data Dispatching Strategies

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

Abstract

Besides Microsoft's Distributed Component Object Model (DCOM) [12] and Sun's Java-based Remote Method Invocation (RMI) [20], OMG's Common Object Request Broker Architecture (CORBA) [14] is now one of the most important middleware architecture standards in the field of object-oriented and distributed client-server application systems. However, the current CORBA standard lacks some comfortable facilities for enabling group communication and parallel processing. For parallel programming purposes, it is still necessary to fall back on specialized achievements such as the Parallel Virtual Machine (PVM) [5] and the Message Passing Interface (MPI) [13], even if the problem to be solved does not justify the overhead of a fully-blown solution for parallel programming.

As an approach to this problem, we have designed a simple, but very flexible Object Group Service for CORBA which can be used to manage data dispatch to several servers in many different ways and, thus, facilitates the development of CORBA-based, distributed, and parallel software applications. Before we describe the basic architecture of this new service, we give a short overview of some of the basic problems that have to be solved in parallel programming concerning asynchronous communication and non-blocking clients. We analyze the possibilities CORBA provides to deal with these problems, and we identify the concepts that are helpful for the development of parallel applications. Afterwards, the Object Group Service and the different dispatching strategies supported by the service are presented and explained in detail.

1. Introduction

Today, the development of software applications based on parallel processing usually requires the use of specialized and often complex tools and message passing libraries such as the Parallel Virtual Machine (PVM) or the Message Passing Interface (MPI). As modern object-oriented programming environments involving programming languages such as Java and middleware architectures such as CORBA become more widespread, it is worthwhile to search for simple solutions in that context. The emergence of products which aim at enabling parallel and distributed programming with Java, such as JPVM [8], jPVM (formerly JavaPVM) [9], JavaMPI [7], HPJava [6], or DOGMA [1], gives evidence of the need for this kind of solutions in the Java world. 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 [4] and [21].

However, in the CORBA environment there is still a lack of simple language- and platform-independent approaches to the problems of parallel processing and group communication. Therefore, our focus in this paper lies on CORBA as a potential middleware foundation for applications based on distributed parallel programming. We present our design of an Object Group Service (OGS) for CORBA, which can be used for forking computational tasks and delegating the tasks to several servers where they are performed in parallel. The design is simple but at the same time very flexible and supports different data dispatching strategies to meet a great variety of application needs.

Coming back to the suitability of CORBA as a foundation for distributed, parallel applications, we first have to analyze some basic CORBA characteristics. In a typical parallel processing scenario, there might be a "master" process (client) residing on one computer in the network which dispatches tasks that are to be executed in parallel by several "worker" processes (servers) living on different computers. In order to be able to efficiently use concurrently active workers, it can be desirable to find a way of triggering the workers via CORBA asynchronously to prevent the master from blocking. Blocking occurs when a client communicates synchronously with the server and has to wait for the server's result or acknowledgement. Since a single-threaded, blocked client cannot perform any computations (so that valuable processing time will unnecessarily be lost if the client does not depend on the immediate return of the server's result), we have to make sure that this situation does not occur. This is even more important in the case of communication errors or errors on the server which lead to the consequence that there will be no acknowledgement to the client at all, thus keeping the client waiting for an indefinite period of time (or, to be more precise, a pre-specified timeout period).

Unfortunately, the basic communication model of CORBA is synchronous and blocking. 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 the problem on the programming language level in Java.

2. Three Ways of Realizing Asynchronous Operation Calls in CORBA

Analyzing the standard, there are basically three possibilities to be found how asynchronous and/or non-blocking calls in CORBA can be attained (cf. [15]):

1. by using the keyword **`oneway`** of the Interface Definition Language (IDL),
2. by using the Dynamic Invocation Interface (DII), and
3. by using the CORBA Event Service.

The IDL keyword **`oneway`** serves for specifying the direction of the information flow between client and server. If a server operation is marked as **`oneway`**, it can only receive information from the client (the call itself and its arguments) but will not return any acknowledgement or other information. Therefore, only "**`in`**"-parameters are allowed for **`oneway`**-operations, 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.

Another way of reaching our goal is to employ the Dynamic Invocation Interface (DII) of CORBA. Using the DII for asynchronous requests requires the call of a specialized request operation named **`send_deferred()`**. Its peer is an operation named **`get_response()`** which has to be invoked at any time anywhere in the program code in order to obtain the results of the dynamic request. However, even if the invocation of **`send_deferred()`** is asynchronous, the blocking of the client cannot always be avoided: if the server has not finished its computation when **`get_response()`** is called, the client will inevitably have to pause. If blocking is not tolerable, the developer will be forced to fall back on the multithreading features provided by the particular programming language used for the implementation of the client. A combination of the two possibilities described so far is a "dynamic" **`oneway`**-invocation based on operation **`send_oneway()`**. Because of the **`oneway`**-semantics of this approach, no blocking of the client can happen, since there are no calls to operation **`get_response()`** at all.

The last of the built-in features of CORBA which can be used to achieve asynchronous and/or non-blocking communication is represented by the CORBA Event Service. However, being part of the Common Object Services Specification (COSS), it does not belong to the essential CORBA core. The "behavior" of the Event Service can be described with the help of the well-known "publish/subscribe"-pattern. There are suppliers and consumers who make use of the Event Service by communicating over a so-called Event Channel in order to perform their tasks. The CORBA Event Service supports different pure or hybrid Pull and Push Models to allow different event propagation strategies.

Yet, the Event Service shows a number of considerable drawbacks with respect to some typical requirements that can be found in application development projects:

- the actual deliverance of an event to an event consumer cannot be guaranteed by the service,
- event filters are not supported, so that large numbers of events are propagated to consumers who are not responsible for the processing of those particular events,
- no additional information such as the current number of event consumers etc. is provided.

One attempt to handle these disadvantages is OMG's Notification Service, which we are not going to describe here. In [17] and [18], further techniques for synchronous and asynchronous communication in CORBA can be found, focusing on "CORBA messaging".

3. Multithreading on the Language Level Against Client Blocking

Because of the restrictions connected with asynchronous operation calls in CORBA, it is usually more advantageous to rely on synchronous communication means. However, as we said before, blocking of the master is a major hurdle if the developer intends to use parallel processing efficiently. So, if asynchronous invocation techniques are not to be used, it will be necessary to find a different way of avoiding blocking. 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 the developer chooses C, C++ or Java, then blocking can be avoided by aptly employing multithreading techniques, although this might lead to new problems, e.g. restrictions regarding portability in the case of C and C++. In any case, following the language-level approach means that development efforts will increase and readability of the source code will decrease.

4. Three Ways of Realizing Group Communication in CORBA

In the CORBA environment, group communication features can possibly be realized on different abstraction levels. For example, group communication could be based on

1. the GIOP protocol level,
2. the IDL level, or
3. the application/service level.

An example of the use of the first alternative, together with a multicast group communication engine, can be found in [10] and [11]. It requires a specific adaptation of the communication protocol. An important disadvantage of this approach is the lacking compatibility with the existing CORBA standard: applications which make use of this protocol cannot communicate directly with applications based on message exchange via IIOP.

The second alternative presupposes an extension to the IDL. For this purpose, we could introduce a new IDL keyword. According to the current OMG specification, operations have to be declared conforming to the following production rule:

```
<op_dcl>      ::= [<op_attribute>]
        <op_type_spec> <identifier>
        <parameter_dcls>
        [<raises_expr>]
        [<context_expr>]
```

At present, **`<op_attribute>`** can only be replaced by the keyword **`oneway`**. Therefore, an extension with a new keyword such as **`multicast`** would be needed. In order to avoid blocking, the semantics of **`multicast`** have to imply the **`oneway`** semantics, so that the two keywords should not be used in combination. Like the first alternative, this approach would have a negative effect on the compliance with the CORBA standard. Because of the extension, not every IDL compiler would be capable of compiling the IDL files. In the case of Java, using the generated stubs and skeletons would not be possible with every ORB available.

The last approach seems to be the most flexible one, because it works with any CORBA-conforming ORB product and does not require any modification to those products. In the following sections, we are going to present our prototypical application/service level solution to the problem of group communication and parallel processing.

5. An Object Group Service (OGS) for Parallel Processing

In order to support the design of applications using parallel processing already on the IDL level and to solve the problem of lacking features for group communication in CORBA, we developed our own Object Group Service (OGS).
For the design of our OGS, we had the following primary goals in mind:

- 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 improve network traffic (compared to sending the complete data to each worker).

The notion of a CORBA-based Object Group Service was first introduced by Felber ([2], [3]). However, while the OGS in Felber's approach serves for supporting replication scenarios, the application purpose of our OGS is totally different, because it aims at facilitating the parallel processing of CORBA operation calls. With the help of the OGS, a request issued by a master can be propagated to any number of workers, which process the data transferred to them independently of each other and in parallel.

The IDL listing on the next page shows the core design of the OGS on the interface level. A client that wants to use the dispatching services of the OGS needs to implement an interface **`Master`**, containing only one single operation **`receive()`**, which gets information issued by a server (a worker) registered to the OGS. Similarly, the worker 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 master. The master's IOR is needed by the worker in order to be able to send the result back to the master via a callback. An interaction begins when the master invokes operation **`send()`** on a particular group of workers, thus transmitting the message to be executed as well as its own IOR to the group. Next, the message will be propagated from the group object to its members, the workers. Finally, the workers will inform the master of the result of their computations by calling its operation **`receive()`**.

Dispatching a call issued by the master to all members of a group of workers is the responsibility of an implementation realizing the IDL interface **`Group`**. In addition, this interface specifies functionality for attaching and detaching workers to/from a group. If a worker tries to register itself twice, an **`Already-Registered`** exception will occur. Vice versa, an attempt to detach a worker which is not registered raises 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.

The purpose of the **`GroupManager`** interface is to facilitate the administration of more than one group of workers. To this end, it provides lifecycle and query operations such as **`create()`** for the creation of a new group, **`list()`** to show the current set of groups, **`resolve()`** to determine a reference to a specific group, and **`destroy()`** for the deletion of a group.

Operation **`create()`** will raise an **`AlreadyExists`** exception in case a group with the same name 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 **`destroy()`** will lead to a **`NotAvailable`** exception.

The UML Deployment Diagram in Fig. 1 shows a physical model corresponding to our implementation of the OGS at the instance level. It depicts a snapshot of a particular point in time during the execution of a system implementing the OGS.

The 3D boxes in the diagram symbolize different computational nodes, such as the computer on which the master is running, the network etc. The computational nodes accommodate the UML components (model elements representing the corresponding CORBA objects) implementing master, workers etc. Connected to these components are "lollipop" symbols, explicitly representing their IDL interfaces. Channels of information flow and connections on the hardware level are shown as solid lines between the computational nodes. The dashed lines express dependencies between the different components, e.g. a component uses an interface of another component. For example, since the **`master`** component instance needs to be able to get a list of available groups, it must be able to call operation **`list()`** on the **`grpman`** component instance and, thus, depends on the **`GroupManager`** interface of the **`:Group-Manager`** component.

The following IDL interface gives an overview of the architectural design of the Object Group Service:

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

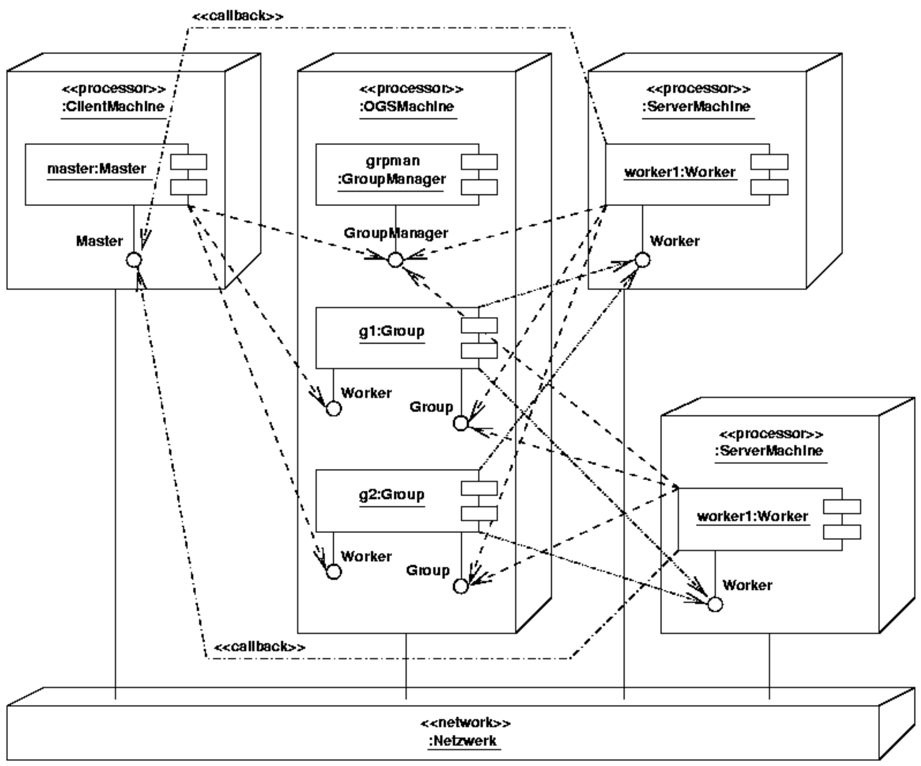

Fig. 1: Deployment Diagram of the OGS Example Scenario

```
  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;
  };

  typedef DataStructure ds;

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

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

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

  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);
  };
};
```

A **`DataStructure`** contains the data that has to be distributed as well as information about the dispatching strategy that has been chosen. While the data itself is stored in an element called **`data`**, the strategy for data distribution is put in element **`dds`**.

Both the **`result`** parameter of operation **`receive()`** and the **`data`** element inside the **`ds`** data structure 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.

To illustrate the functionality of the OGS further, we present an example scenario of the use of the OGS for parallel programming. 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. Finally, the master requests a list of the existing groups, sends a message to groups **`g1`** and **`g2`** and eventually receives the result of its call via a callback, i.e., the workers themselves invoke the master's operation **`receive()`**.

The UML sequence diagram in Fig. 2 shows the dynamic flow of messages in the scenario described above. It 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 as 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. 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.

`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 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

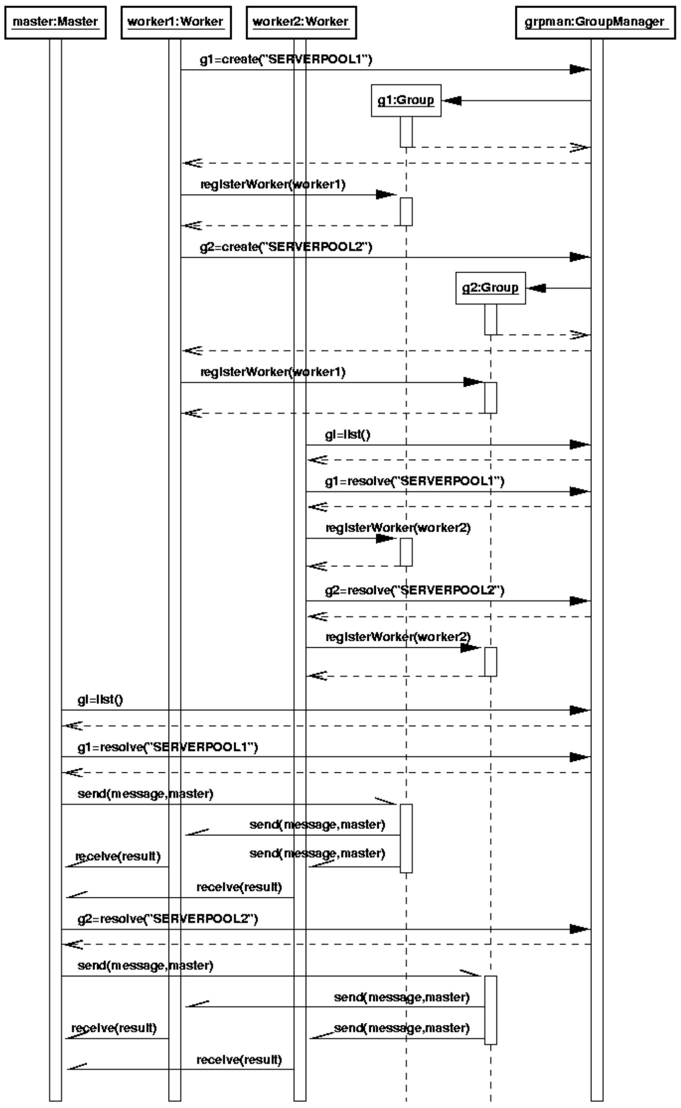

Fig. 2: Sequence Diagram of an Example Scenario

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: {10, 20, 30, 40, 50}. 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	10 40	20 50	30
DIFFERENT_SIZE { 2, 1, 7 }	10 20	30	40 50
SAME_SIZE { 3 }	10 20 30	40 50	—
ANYTHING or default	10 20 30 40 50	10 20 30 40 50	10 20 30 40 50

Table 1: Distribution of Data Resulting from Different Dispatching Strategies

7. Applying the OGS—Two Examples

The UML sequence diagram in section 5 helped to explain the basic dynamic communication behavior of a parallel system using the OGS, and the description of the different dispatching strategies supported by the OGS elucidated its functionality. In this section, we present some code snippets of typical master and worker implementations in Java which illustrate how to use the OGS.

In general, 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()** 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 determines 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(
  DataStructure message,
  Master master)
{
    // extract data
    l_arrayHelper data =
      new l_arrayHelper();
    int[] vec =
      data.extract(message.data);

    // 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 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 **3**, **0**, **2**, i.e., the first worker will get three, the second zero, and the third worker will get two 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, 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.

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

int[] vec1 = { 1, 2, 3, 4, 5,
               6, 7, 8, 9 };

l_arrayHelper data_helper =
  new l_arrayHelper();
data_helper.insert(data,vec1);

int[] lenSeq = new int[3];
lenSeq[0]=3;
lenSeq[1]=0;
lenSeq[2]=2;

DataStructure message =
  new DataStructure(
    DataDispatchStrategy.DIFFERENT_SIZE,
    lenSeq,0,any);

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_helper.insert(data,vec2);
message = new DataStructure(
  DataDispatchStrategy.SAME_SIZE,
  lenSeq,3,any);

g2.send(message, master);
```

8. Advantages of the OGS Over CORBA's Event Service

A comparison of our Object Group Service and the CORBA Event Service points out several advantages of the OGS. Its design and functionality is not restricted to broadcast communication, i.e., calls from the master being propagated to all the workers, but also provides the following benefits:

- 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. Therefore, our solution is less resource consuming, since only one OGS 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.

9. Summary and Outlook

In this paper, we have pointed out that in many cases it might be desirable to use a simple CORBA-based solution instead of proprietary approaches to support parallel processing and group communication.

However, CORBA is lacking satisfactory features supporting the requirements of parallel and/or group communication-based applications. We have identified an important prerequisite for this kind of applications, namely asynchronous communication, or, more precisely, the possibility to avoid blocking clients in order to make efficient use of parallel processing. Several approaches to the achievement of this goal with the means available in the CORBA world have been identified and discussed.

For example, specifying that a particular operation can be called asynchronously by using the keyword **`oneway`** has the advantage, that the solution is independent of the programming language in use, so that non-blocking, asynchronous calls can be made from COBOL, Ada, Smalltalk, C++, and Java code, likewise. One current problem we encountered was the failure of **`oneway`** calls performed on the basis of OmniBroker and OAK on Linux, whereas other combinations of ORB products and platforms worked quite well.

Solutions to the problem of blocking clients on the programming language level, using multithreading, are especially simple and comfortable in the case of Java. Since the code remains portable, multithreaded Java code can also be a satisfactory approach. In the case of other languages, solutions on the IDL level are usually more suitable.

The last alternative we have presented is OMG's Event Service, which is a quite powerful tool for enabling asynchronous communication. On the other hand, it does not support multicast communication sufficiently and provides too much overhead for use in small and simple applications.

Since multicast or group communication is needed for our purposes, we have mentioned three potential approaches. However, two of them require the modification of existing protocols or standards, so that we have decided to find a solution on the application/service level by designing our Object Group Service.

Some of the major advantages of the Object Group Service over the Event Service are mentioned in section 8. The complexity of the code fragments needed in applications that use the OGS is very low, so the application structures remain clear and understandable and the programs can thus easily be maintained. The OGS is a flexible service for dispatching any type of data to any number of workers computing their partial results in parallel. Besides the simplest dispatching strategy, i.e. transmitting the complete data to each worker, the OGS supports several other strategies which guarantee that each data element is sent across the network to only one worker at most, and also enable different kinds of computing algorithms to be performed on the workers.

At present, we are implementing a Join Service for recollecting the workers' results, packing them, and sending the complete end result back to the master. With the Join Service, no more callbacks from the single workers to the master take place, so that the master's **`receive()`** operation is only called once. Thus, the master's responsibilities concerning the processing of the results are decreased, i.e. the master can be simpler, which compensates the disadvantage of the indirect communication channel.

References

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

[2] 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

[3] 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

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

[5] 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

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

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

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

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

[10] L. E. Moser, P. M. Melliar-Smith, R. Koch, K. Berket (1999): "A Group Communication Protocol for CORBA", in: Proceedings of the International Conference of Parallel Processing Workshops, 21-24 Sept. 1999, Aizu-Wakamatsu, Japan, pp. 30-36, http://www.beta.ece.ucsb.edu/~ruppert/pdf/iwgc99.pdf

[11] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, R. R. Koch, K. Berket (1999): "Multicast Group Communication for CORBA", in: Proceedings of the International Symposium on Distributed Objects and Applications (DOA), 5-6 Sept. 1999, Edinburgh, Scotland, pp. 98-107, http://www.beta.ece.ucsb.edu/~ruppert/doa99.pdf

[12] 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

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

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

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

[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