Handling Exceptions Between Parallel Objects[⋆]

Christian Pérez, André Ribes, and Thierry Priol

IRISA/INRIA, Campus de Beaulieu 35042 Rennes Cedex, France
Tel:+33 2 99 84 74 13, Fax:+33 2 99 84 71 71
{Christian.Perez,Andre.Ribes,Thierry.Priol}@irisa.fr

Abstract. Scientific computing is evolving from parallel processing to distributed computing with the availability of new computing infrastructures such as computational grids. We investigate the design of a component model for the Grid aiming at combining both parallel and distributed processing in a seamless way. Our approach is to extend component models with the concept of parallel components. At runtime, most component models rely on a distributed object model. In this paper, we study an exception handling mechanism suitable for dealing with parallel entities, such as parallel objects, that appear as collections of identical objects but acting as a single object from the programmer's viewpoint.

1 Introduction

Scientific computing has become a fundamental approach to model and to simulate complex phenomena. With the availability of parallel machines in the late 1980's, a strong emphasis was made to design algorithms and applications for scientific computing suitable for such machines. However, nowadays, science or engineering applications, such as multiphysics simulations, require computing resources that exceed by far what can be provided by a single parallel machine. Moreover, complex products (such as cars, aircrafts, etc.) are not designed by a single company, but by several of them, which are often reluctant to grant access to the source of their tools. From these constraints, it is clear that distributed *and* parallel processing cannot be avoided to manage such applications. More precisely parallel simulation codes will have to be interconnected through a specific middleware to allow a distributed execution, thus complying with localization constraints and/or availability of computing resources. Programming a computing resource, such as a Grid infrastructure, that has both dimensions, parallel and distributed processing, is challenging.

The objective of our research activities is to conciliate these two technologies in a seamless way achieving both performance and transparency. More specifically, we aim at providing a programming model based on the use of distributed software components. Since most component models are based on distributed objects, much of our work focuses on extending a distributed object model. This

[⋆] This work was supported by the Incentive Concerted Action "GRID" (ACI GRID) of the French Ministry of Research.

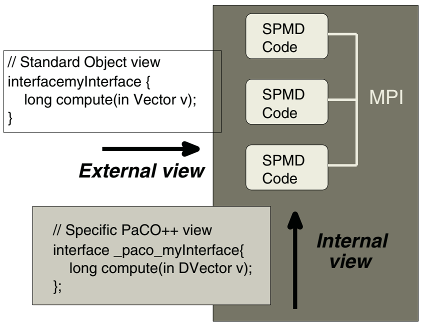

Fig. 1. A PaCO++ parallel object.

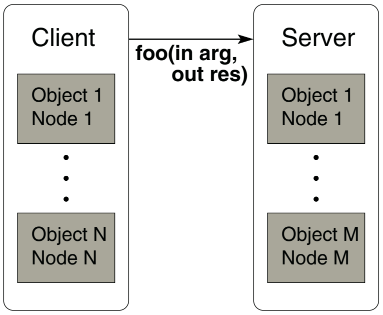

Fig. 2. Example of a distributed application.

work specifically aims at handling exceptions in the presence of a collection of distributed objects, called parallel objects in the remaining of the paper. A collection of distributed objects is the basic principle we introduced into Corba for the encapsulation of parallel codes that rely on a SPMD execution model.

Section 2 introduces PaCO++ which implements the extension to the Corba distributed object model for managing collections of identical objects. In Section 3, we describe our proposed model to manage exceptions in the presence of collections of objects. Related works aiming at handling exceptions in the context of concurrent or parallel systems are presented in Section 5. Section 4 gives some technical details about the implementation of the exception handling mechanism and Section 6 concludes the paper.

2 A Parallel Object Model: PaCO++

From the early 90's, several projects were set up to build distributed object middleware systems. However, most of these projects did not provide an efficient and adequate support for handling parallel codes. Research activities started in the late 1990's to support parallelism such as Corba [5, 9, 6, 8]. All of them introduce the concept of parallel object. A parallel object is an entity that behaves like a regular object but whose implementation is parallel. Hence, a parallel object can be referenced and methods can be invoked on it as a single entity. Some of them also support data redistribution.

This section briefly describes PaCO++ [8], our research platform, so as to let Section 4 illustrates how parallel exceptions might be implemented. PaCO++ is the continuation of PaCO [9]. It is a *portable* extension to Corba that is intended to be the foundation of GridCCM [7], our parallel extension of the Corba Component Model. PaCO++ defines a parallel object as a collection of identical Corba objects whose execution model is Spmd. A parallel object is implemented by a coordinated set of standard Corba objects. As shown in Figure 1, a PaCO++ parallel object provides a standard external interface which is mapped to an internal interface. An invocation on a parallel object results in the simultaneous invocation of the corresponding operation of the internal interface on all the objects being part of the parallel object. Special attention has been devoted to the support of distributed arguments. The arguments can be distributed at both the client and the server sides. PaCO++ provides mechanisms (static and dynamic) to specify the data distribution on both sides. As the two data distributions may be different, data redistribution may be required during the communication between the client and the server.

3 Managing Exceptions Within Parallel Objects

The problem addressed in this paper is the management of exceptions in the presence of parallel objects such as those provided by PaCO++ (but it is not limited to them). If we consider a distributed application made of two parallel objects (a client and a server) like in Figure 2, it may happen that an operation invocation of the client on the server side raises an exception. However, since the server is a collection of objects, several scenarios may happen depending on which objects have raised an exception at the server side. One or several objects may raise exceptions with different values associated with them. It is thus necessary to define a model for exception management that gives a coherent view of the exceptions raised by the server to all the objects of a parallel client.

Scenarios. We have identified four scenarios that need to be handled. The simplest one is a *single exception* where only one object of the collection throws an exception. The *multiple exception* occurs when several objects of the server throw potentially different exceptions. A Spmd *exception* requires that all objects at the server side coherently throw the same type of exception. Last, a *chain of exceptions* occurs when the server calls a method on another remote object and this second object throws an exception.

Definition of a parallel exception. We define a parallel exception as the collection of exceptions thrown by one or several nodes of a method of a parallel object. We define an Spmd exception as an exception declared as Spmd by the parallel object. From an execution point of view, it is a parallel exception which is made of exceptions of identical type which have been coordinately raised by the all nodes of a parallel object.

3.1 Motivating Application

Let us introduce the EPSN project [1] as an application that motivates this work. Its goal is to analyze, design and develop a software environment for steering distributed numerical simulations from the visualization application. As shown in Figure 3, it adds a level of constraints which stem from the human interaction loop: data have to be extracted from the application and sent to a visualization machine but also the user actions have to be sent back to the application. As a user can connect to and disconnect from a running application from *a priori* any

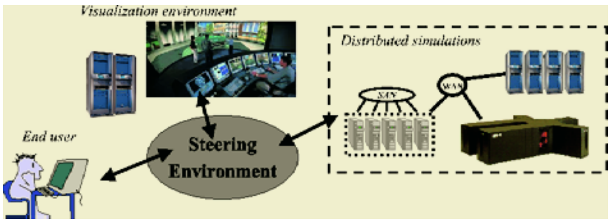

Fig. 3. Overview of the EPSN project.

machine, a distributed-oriented middleware appears well-suited to handle such interactions.

When the visualization application requests data, the simulation code might throw an exception because of memory limitation for example. Since the visualization code and the simulation code are parallel, the middleware has to manage the "parallel" exception. In particular, each node of the visualization application has to receive the exception. If the visualization tool can handle partial images, the exception may be able to carry a partial image. In the same time, it is possible to have several visualization clients for one simulation. Some of them may be able to manage partial data and some not. Hence, different exceptions need to be returned to the different clients.

3.2 The Three Types of Exceptions

Only three types of exception seem to be sufficient to handle all the previous scenarios. A *simple exception* type is just a plain exception. It can be obtained either when only an object of the server throws an exception or when an SPMD exception is raised. Section 3.4 will detail this. An *aggregated exception* type is composed of all the exceptions raised by the server objects. For example, it occurs when several objects of the server throw different exceptions. These exceptions have different meanings and cannot be grouped in a *simple* exception. A *complex* exception is composed of an *aggregated* exception with uncomplete data. Uncomplete data is the current value of the out arguments of the operation that generates the exception. Some restrictions may be applied to uncomplete data as described in Section 3.5.

3.3 Client Side

A client may not be aware of the proposed new types of exceptions or may not be able to catch them. Therefore, the model must allow a client to choose the kind of exceptions it wants to handle. The model defines three exception levels that correspond to the three types of exception defined in Section 3.2. By default, a client is assumed to only support simple exceptions (*simple level*). This level is the default level because of the legacy clients which are unaware of parallel objects. The second level which is the *aggregated level* is used by clients which are able to catch exceptions composed of several regular exceptions. The last level, *complex level*, indicates that the client may catch exceptions composed of an aggregated exception and uncomplete data. These levels form a hierarchy. A client that handles *aggregated exceptions* implicitly handles *simple exceptions*. Similarly, a client handling *complex exceptions* is assumed to also handle *simple* and *aggregated exceptions*.

3.4 Server Side

For each exception that a method of a parallel object may throw, additional information may be needed for a correct exception handling. First, SPMD exceptions have to be declared as such and data distribution information need to be specified for the distributed data the exception may contain (see Section 3.5). Second, priorities may be attached to exceptions. These priorities are fixed by the application designer (server side) and are not seen by a client. They are primarily used to decide which exception will be thrown to a client that only supports the *simple level* when several unrelated exceptions are raised. Third, the out arguments of a method that need to be returned as uncomplete data also have to be declared.

To determine the type of exceptions the server needs to send to a client, the server needs to know the level of exception the client supports. There are four cases depending on the type of exception raised by the parallel server and the level of exception supported by the client. First, if only one node of the server throws an exception, the server may safely send this exception as the client at least supports the *simple* exception type. Each node of a parallel client will receive this exception. Second, if every node of the server throws the same type of exception and the exception is defined as an SPMD exception, the server also applies the same algorithm as in the first case but also adds the management of distributed data. A sequential client is seen as a parallel client with one node. Third, if several nodes of the server throw exceptions and the client supports the *aggregated* level, the server sends an *aggregated* exception composed of all the exceptions raised. The implementation may send additional information like the object identifier where the exception has occurred. If the client only supports *simple* exceptions, the server throws the exception with the highest priority. Fourth, if the client supports the *complex* case and some out arguments are marked as valid for uncomplete data, a *complex* exception is returned.

3.5 Managing Data Within a Parallel Exception

Managing distributed data within a SPMD *exception.* An SPMD exception may contain data declared as distributed. When such an exception is raised, the runtime system may need to redistribute the data to the client (e.g., the client may be composed of a different number of nodes than the server). Since an exception can be seen as a structure with different data fields, there is no difference with the management of distributed arguments. As the model is not designed for a particular distributed object middleware, it just specifies that an implementation has to use the same mechanism as the out argument management in the parallel distributed middleware where it is implemented. Some implementation hints for PaCO++ are given in Section 4.

Uncomplete data. Uncomplete data can only be encapsulated into complex exceptions. If the client does not support this exception type, the server cannot send uncomplete data. When the runtime system knows all the raised exceptions, it can determine which out arguments need to be sent to the client. Then, a complex exception made of several exceptions and the valid out arguments is created. Otherwise, a complex exception without out arguments are returned.

3.6 Managing Exception Chains

For example, an exception chain occurs when a client invokes a method A on a first server which invokes a method B on a second server that raised an exception. In the general case, an exception chain occurs when there are several – at least one – servers between the client which catches the exception, and the server which raises the exception. Ideally, we would like to send an aggregated exception combining the server exception and an information that this exception was thrown by an another server. This exception can be sent only if the client and the server support aggregated exceptions and that all intermediate servers have declared the exception type. Otherwise, either an unknown exception or a model specific exception like *chain of exception* should be raised.

4 Implementation Strategy

This section provides some hints on how to implement our model into PaCO++.

PaCO++ adds a layer between the user code and the Corba code (e.g., stubs and the skeletons generated by an Idl compiler). This layer is generated thanks to the description of the distributed service (Idl) and the parallel description file specific to PaCO++, which describes the parallel methods and the distributed arguments of these methods. The main implementation principle is to extend this file with exception related features. For each exception defined in the Idl file for a parallel method, the designer indicates the priority of the exception, whether this exception contains distributed data, etc. For each parallel method, two new exceptions are added into the generated Idl file: one for the aggregation case and one for the complex case.

The client and the server usually need to configure the PaCO++ layer. For example, PaCO++ layers need to obtain a reference to the distribution libraries. The client and server are configured using their contexts. These contexts need to be extended with information related to exceptions.

Each parallel method is mapped to two asynchronous methods, one for each direction. Thus, for the PaCO++ layer, the exception is just an another kind of result. We only have to generate the code for these methods. The management

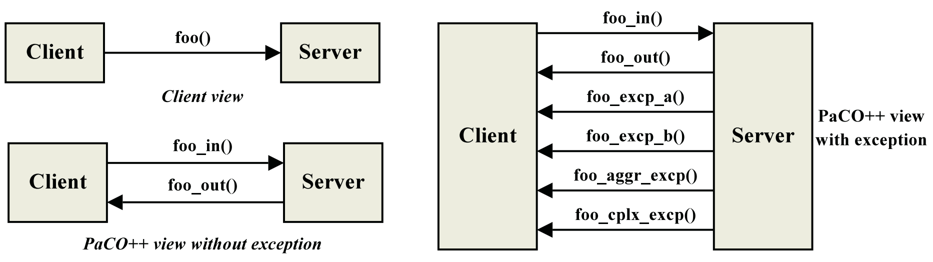

Fig. 4. Exception-related methods to transfer exceptions in a PaCO++ parallel object.

of data distribution is the same as for a regular invocation. Figure 4 shows these new methods for a parallel method `foo` that may throw exceptions of types `A` and `B`. The server code contains a barrier to synchronize all the objects before returning the results to the client. If an object throws an exception, the information is broadcast so that all objects have the same view of the exception state. After the barrier, the runtime system first checks if there is an exception; if not, the runtime system performs the normal code. So, a boolean test is added to the normal code. Thus, the overhead to supporting parallel exceptions in PaCO++ should be negligible for normal invocations.

5 Related Work

Several solutions have been proposed to support exceptions in parallel code. Some works [3, 10, 4] try to handle multiple exceptions within parallel loops. In [3], Modula-3 is extended so that if one or more exceptions of the same type are thrown within a parallel loop, the loop throws only one exception. But, if there are two or more different exception types, the runtime system sends an error. In [10], it is proposed to extend Java to have parallel loops with exception support. Each exception is stored into an aggregated exception that is thrown to the caller. The caller may retrieve some uncomplete data. In [4], when a node of the parallel loop throws an exception and another node communicates with it, the first node sends its exception to the other nodes. This exception is called a *global* exception. For the caller of the parallel loop, the runtime system generates a *concerted* exception from all the exceptions thrown.

In [2], it is proposed to manage exceptions between groups of distributed objects. Each group has one or more gates that enable to communicate with other gates. The invocations are asynchronous. The client has to call a method to get the result of the operation. In the server, if one object throws an exception, this exception is saved on the gate and the client will be aware of this exception when the synchronization method is called. Finally, Xu et al. [11] present a solution to manage exceptions between different processes that take part in the same *action*. They describe how to rollback this action when an exception occurs.

6 Conclusion

Handling exceptions is of prime importance when designing an object or component model, especially when it targets Grid infrastructures that combine both parallel and distributed aspects. The main contribution of this paper is to define the concept of parallel and SPMD exceptions for parallel objects implemented as a collection of identical sequential objects. The proposed model is able to manage all identified scenarios including concurrent uncoordinated exceptions raised by a parallel object. It also defines aggregated exceptions and complex exceptions with uncomplete data by integrating previous works on exceptions within parallel loops. Moreover, our proposed model gives a solution to hande distributed data for SPMD exceptions. We are currently implementing the proposed model in PACO++. As outlined in this paper, the implementation appears straightforward and should exhibit a negligible overhead for normal operation. We are also working on defining a mathematical model of parallel exceptions.

References

1. The EPSN Project, http://www.labri.fr/Recherche/PARADIS/epsn/
2. Chris Exton and Ivan Rayner. Exception semantics in a parallel distributed object oriented environment. In *Proc. of the 21 International Conference TOOLS Pacific*, page 51, Melbourne, Australia, November 1996.
3. Ernst A. Heinz. Sequential and parallel exception handling in modula-3*. In P. Schulthess, editor, *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, pages 31–49, Ulm, Germany, September 1994.
4. Valérie Issarny. An exception handling model for parallel programming and its verification. In *Proc. of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 92–100, New Orleans, Louisiana, USA, December 1991.
5. K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *Supercomputing'97.* ACM/IEEE, November 1997.
6. Object Management Group. Data parallel CORBA, November 2001. ptc/01-11-09.
7. Christian Pérez, Thierry Priol, and André Ribes. A parallel CORBA component model for numerical code coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003.
8. Christian Pérez, Thierry Priol, and André Ribes. PaCO++: A parallel object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems Minitrack in the Software Technology Track of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, Big Island, Hawaii, USA, January 2004. IEEE Computer Society Press.
9. C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, Redondo Beach, California, USA, August 1999. IEEE.
10. Joel Winstead and David Evans. Structured exception semantics for concurrent loops. In *Fourth Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, Tampa Bay, October 2001.
11. J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. In *IEEE Trans. on Parallel and Distributed Systems. TPDS-11, N 10*, 2000.