Object serialization and remote exception pattern for distributed C++/MPI application

Karol Bańczyk, Tomasz Boiński, and Henryk Krawczyk

Gdańsk University of Technology, Faculty of Electronics, Telecommunication and Informatics, ul. Gabriela Narutowicza 11/12, 80-952 Gdańsk,
kaban|tobo|hkrawk@eti.pg.gda.pl

Abstract. MPI is commonly used standard in development of scientific applications. It focuses on interlanguage operability and is not very well object oriented. The paper proposes a general pattern enabling design of distributed and object oriented applications. It also presents its sample implementations and performance tests.

Key words: MPI, object serialization, remote exception handling

1 Copyrights

Boinski T., Banczyk K., Krawczyk H., Object serialization and remote exception pattern for distributed C++/MPI application, PaCT 2007, LNCS 4671, pp. 188193, 2007. Springer-Verlag Berlin Heidelberg 2007

Full version available on: http://www.springerlink.com/content/2lm75641g0634727/

2 Introduction

MPI[1] is a widely accepted standard for message passing in scientific applications. It focuses on interlanguage compatibility (FORTRAN, C, C++) rather than on leveraging a single language constructs. Nevertheless, in many C++ applications a more object oriented, MPI based network interface (later referred to as connector) would be desireable. Although bindings for C++ were introduced to MPI [1], more sophisticated features are often needed for practical use.

This work focuses on object serialization and remote exception handling. The former is a mechanism for converting objects between their in-memory representations and a stream of bytes. The latter allows us to transmit exceptions occurring in a remote server process to the calling client process. Some example applications are also shown.

The paper consists of five sections: Section 2 define design goals; Section 3 presents the proposed pattern; Sections 4 and 5 provide sample implementations and Section 6 discusses certain experimental results.

3 Design Goals

The connector should provide methods for collective and for point-to-point communication as well the possibility to receive exceptions that occurred remotely. A remote exception should be handled in the same way as any local exception. An application satisfying those features could be implemented without any new communication layer. Most of the design goals could be achieved using either the SR language [5] or its Java based ancestor, JR [6] [4], or else MPI wrappers for Java (like mpiJava [10]).

It is risky to write a sophisticated program in a language, such as SR, which has small community around it and few available libraries. Java serialization mechanisms has negative impact on overall performance (which is confirmed by the below mentioned results). Similarly, the mpiJava, as a wrapper around C, introduces additional overhead. So we decided to create the C++ application and implement simplified versions of suitable Java oriented mechanisms.

4 Architecture of Application Pattern

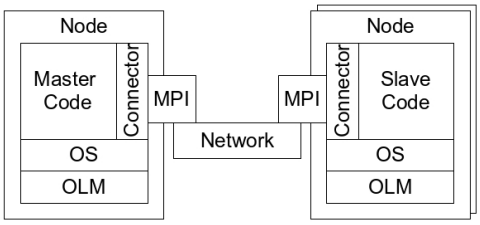

Fig. 1. Application architecture pattern

The assumed architecture is depicted in Fig. 1. with two lower layers: Object Lifecycle Management (OLM) and Object Serialization (OS), both inspired by Java's mechanisms, i.e. reflection and serialization. The former enables class identification and memory management for objects. The letter provides methods for writing and reading objects to/from a stream of integers.

Serialization, though inspired by Java[8], is a simple solution. Per class implementation is needed for each serializable object, no security issues are considered and the serialized stream contains no matadata. This solution requires more development time, but reduced serialization time. Similarly, in Java a per class implementation is also needed, if performance issues are a concern.

The connector transmits serializable Message objects between the nodes. It uses MPI as its underlying network communication library but also Object Serialization with Object Lifecycle Management for converting object messages to/from byte sequences required by MPI.

Every exception occurring during handling procedures is caught by the communication layer, transmitted through the network to the appropriate node and thrown the next time that node invokes a method on the communication layer.

5 Implementation

5.1 Basic classes

In the proposition each serializable class has to be subclass of the IntSerializableObject class and has to implement writeToInts and readFromInts methods. A special Exception class was also defined. Unfortunately, it is impossible to transmit the original exception object itself. The C++ specification [7] states that the memory for the temporary copy of the exception being thrown is allocated in an unspecified way thus allowing each compiler implementation to do it differently. This conflicts with the idea of ObjectFactory and could lead to uncontrollable memory leaks if not handled properly. Both the abovementioned Message class and Exception class needs to be serializable.

Some sublasses of Message are defined: SimpleMessage used for wrapping requests and responses; CarrierMessage employed in transmitting any number of different objects of the IntSerializableObject type in one communication attempt; and ExceptionMessage used for wrapping and transmitting exceptions.

Any of given classes can be further subclassed by any number of more specialized ones to better suite the given solution.

5.2 Serialization

Here is and example ofserialization algorithm: two objects, containing fields f1 and f2, wrapped into CarierMessage, will be serialized in the following way:

1. CarrierMessage class Id is written so that Object Factory will be able to recreate it;
2. CarrierMessage's writeToInts method is invoked in such a way that:
   (a) the object's Id is written so that recipient can deduct meaning of this message, i.e. if it is a message with results or a control message,
   (b) number of contained objects is written (here 2),
   (c) for each of the objects its class Id is written and its writetoInts method is invoked; this method stores Id and fields of that object.

Then, the message is being send and after receiving the serialized object it is recreated as follow:

1. Class Id is read and used for recreating the object, CarrierMessage in this case;
2. CarrierMessage readFromInts method is executed; this method:
   (a) restores value of it's Id,
   (b) reads number of contained objects,
   (c) each contained object is being recreated and it's readFromInts method is employed.

5.3 Remote Exceptions

Exceptions were added to MPI C++ bindings. However, they only apply to MPI communication operations so a more general solution for user aplication exceptions is needed.

Fig. 2 presents the algorithm for remote exception handling. When an exception on a remote node occurs and cannot be handled locally on that node, it is caught and wrapped into an ExceptionMessage object. That object is serializable and thus can be transmitted through the network. After that it is deserialized and thrown again on the target node. Later, on a proper solution for the problem, it can be transmitted to the node where the exception originated.

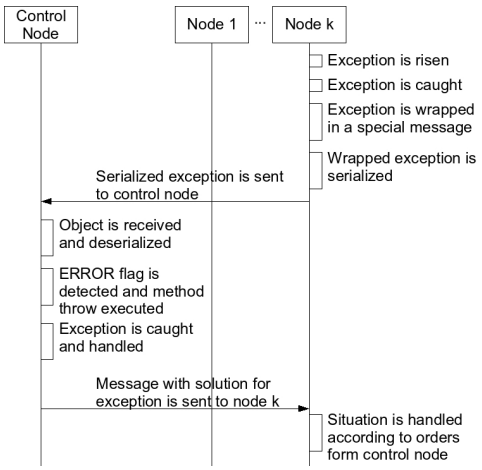

Fig. 2. Remote exception handling algorithm

6 Application Examples

6.1 Assertions

Assertions are a well-known method for finding bugs in software[2]. If a condition is not fulfilled on a single node the standard C++ assert macro silently terminates, leaving the other nodes completely unaware of that fact.

We created our own version called xassert. When the assertion fails on one node, the process throws an Exception. It reaches the master node and causes a standard fatal error handling procedure that can, for example, send all the slaves a termination message and gracefully shut down the whole system.

6.2 Exceptions Thrown In a Slave Node

We developed a master-slave applications designed to solve timetabling problem [3]. Exceptions thrown within slave nodes may sometimes be handled locally, like buffer overflow errors, otherwise have to be passed to master. When, for example, a slave node cannot generate random individuals for given input data, an exception needs to be passed to the master node, where the user is notified about the problem and can apply the solution for it. This approach can be used in any type of client/server approach.

7 Test Results

This section shows the results proving serialization's efficiency. All presented tests were performed on a 10 node cluster. Each node consists of 4 Intel Xeon CPU 2.80GHz and has 4GB of RAM. The nodes are connected via gigabit ethernet network. Measured latency was around 100 microseconds.

7.1 Serialization's Overhead

Table 1. compares time (in microseconds) of preparation of MPI message buffer with serialized objects by means of the proposed serialization mechanisms to writing to a raw integer buffer. For this test we have created a serializable class containing a vector of integers of agiven length. No actual sending is performed.

Table 1. Serialization time in microseconds

Vector size	**With serialization**	**Without serialization**	**With/Without ser.**
1	1.62	0.20	8.10
10	2.74	0.21	13.05
100	4.84	0.40	12.10
1000	10.46	1.37	7.64
		Average:	10.22

The average serialization is 10.22 times slower than writing data directly to integer vector. Nevertheless, it is by one level of magnitude shorter than the latency time, and it simplifies and structurizes the code, which makes the performance loss is acceptable.

7.2 Comparison to Java

Java offers very good methods of serialization and remote exception handling. These methods, however, introduce additional overhead both in terms of needed

time and size of the result. For this test, a simplified connector was implemented in Java. Also two realizations of serialization were provided: normal (standard java.util.ArrayList class with standard serialization) and optimized (using com.sosnoski.util.array.IntArray [9], an array optimized for storing integers and custom read and write methods).

Table 2. Java and C++ serialization comparison

Vector size	**Standard Java**	**Optimized Java**	**C++**
10	32.00	25.00	2.74
100	107.00	16.00	4.84
1000	1180.00	43.00	10.46

The results are presented in Table 2. (in microseconds). Java serialization is slower than the proposed C++ implementation, especially when using standard Java classes. Although simpler to code, optimized Java serialization is 4 to 10 times slower than C++ implementation. Additionally, C++ application in general has smaller memory requirements.

8 Conclusions

The object serialization presented in the paper proved to be efficient and simple for implementation. All types of objects are being serialized in the same manner thanks to usage of integers as a common way of representing data. It allowed us to transport a wide range of objects between nodes. The presented solution provides full transparency both from object's and application's point of view. In all those aspects it is similar to Java solutions yet faster and simpler.

The proposed remote exception handling, is simple but requires forming a special Message objects for each type of exception sent. Those, however, can be coded once and included into a library for future reuse.

In addition to design goals, introduction of a Connector makes applications, built with this patter in mind, extendable. Communication performance and reliability tuning becomes very easy as only changes to Connector needs to be done.

References

1. Message Passing Interface Forum: MPI-2: Extensions to Message-Passing Interface. Message Passing Interface Forum. (1997)
2. Andrew H., David T.: The Pragmatic Programmer: From Journeyman to Master. Addison Wesley Longman. (2000)
3. Bańczyk, K., Boiński, T., Krawczyk, H.: Parallelisation of genetic algorithms for solving university timetabling problems. IEEE Computer Society, PARELEC 2006. (2006) 325–330

4. Hiu Ning Chan et al.: An Exception Handling Mechanism for the Concurrent Invocation Statement. J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par. (2005) 699–709
5. SR Language: http://www.cs.arizona.edu/sr/
6. JR Language: http://www.cs.ucdavis.edu/ olsson/research/jr/
7. National Committee for Information Technology Standards: International Standard ISO/IEC 14882, Programming Language - C++ Approved by NCITS as an American National Standard, http://www.ncits.org/standards/pr14882.htm.
8. Java Serialization Specification, version 1.5: http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html
9. Sosnoski, D.: Type-Specific Collections Library. http://www.sosnoski.com/opensrc/tclib/index.html
10. mpiJava: http://www.hpjava.org/mpiJava.html