Exceptions in remote procedure calls using C++ template metaprogramming

Matti Rintala¹,*,†,‡

¹ Institute of Software Systems, Tampere University of Technology, P.O. Box 553, FI-33101 Tampere, Finland

SUMMARY

Serialisation of data is needed when information is passed among program entities that have no shared memory. For remote procedure calls, this includes serialisation of exception objects in addition to more traditional parameters and return values. In C++, serialisation of exceptions is more complicated than parameters and return values, since internal copying and passing of exceptions is handled differently from C++ parameters and return values. This article presents a light-weight template metaprogramming based mechanism for passing C++ exceptions in remote procedure calls (RPC), remote method invocations (RMI), and other situations where caller and callee do not have a shared address-space. This mechanism has been implemented and tested in the KC++ concurrent active object system.

KEY WORDS: C++, RPC, RMI, distributed systems, exception handling, templates, serialisation

INTRODUCTION

In many programs, it is quite common to have one process (or some other entity) acting as a server, serving requests from clients. If the server and client do not have shared memory, serialisation (often also called marshalling) of data is needed when the caller and callee communicate with each other, often through some sort of message passing system. This need may arise in a distributed environment with RPC (remote procedure call) or RMI (remote method invocation), or in a parallel or concurrent system where the server and the client are separate processes without shared memory. For the rest of this article, the abbreviation “RPC” is used loosely to refer to all these situations. Serialisation
is included in many programming languages like Java, and is a part of many language independent systems like CORBA, but the C++ language has no built-in support for serialisation.

Exceptions are rapidly becoming the most common error-handling mechanism, which means serialisation mechanisms in RPC should also be able to pass exceptions in addition to parameters and return values. This functionality is still missing from many RPC libraries, forcing programmers to revert to traditional return values for error handling.

Standard C++ passes objects as parameters and return values differently from many other object-oriented languages. In C++, objects are passed by copying them based on the static type defined in the function’s interface, rather than cloning the object based on its dynamic type (Java RMI) or sharing objects without copying them (normal Java). Explicit sharing is of course possible in C++ through separate pointer and reference types.

Copying objects based on their static type allows C++ to optimise the performance of passing parameters and return values when objects are small. It also makes serialisation easier in most cases, since copying is based on the static type of the parameter or return type. However, exception handling in C++ differs from parameters and return values in this respect. C++ exception objects are copied based on their real dynamic type, making serialisation of exceptions more complex, when they are passed across address-space boundaries (the interface has no compile-time knowledge on the type of the exception object).

This article analyses how exception handling in C++ affects serialisation in remote procedure calls. A light-weight template-based solution solving these problems is then presented, and its performance and ease-of-use is analysed. The mechanism described in this article originates from an experimental concurrent active object system KC++ [1].

**PROBLEM ANALYSIS**

From the interface point of view, exceptions are in a certain sense an alternative to return values. They also transfer information to the caller, i.e. the type of the exception and possibly data embedded in the exception object. Serialisation services are needed for parameter and return values passing. Similarly, serialisation is needed for exception support. Exceptions must be handled for both synchronous and asynchronous calls. When these calls occur across address-space boundaries, implementing exception propagation becomes significantly more complex. This paper addresses exception propagation across address-space boundaries.

In addition to RPC, serialisation is routinely needed for persistence services [2]. However, persistence normally deals only with the state of objects. Therefore, exceptions (which are mainly control-flow signals) are not an issue with serialisation libraries aimed for persistence.

The C++ language does not provide support for serialisation. Distributed systems like CORBA solve this problem by providing their own serialisation libraries. However, custom code is needed for serialisation of user-defined classes and data types. Complex frameworks like CORBA offer generality and let systems communicate with each other over machine architecture and programming language boundaries.

However, for many light-weight applications, such frameworks are unnecessarily heavy-weight and complex, and they require extra programmer effort in the form of IDL specifications, etc. For example, a program may just need a few simple distributed remote procedure calls, or it may consist of a few
concurrent processes communicating with each other on the same machine. Such applications tend to use lighter-weight serialisation libraries, usually without separate code-generator programs.

As mentioned, C++ copies objects when they are passed as parameters or return values. This behaviour differs from many other object-oriented languages like Java, where objects are passed by reference and the caller and callee share the same object. However, the copying approach is used in most languages for passing basic types (integers, etc.) by value instead of by reference. The copy semantics is also ideal for RPC and serialisation, since sharing an object is difficult in calls across address-spaces and computers (even Java RMI copies its parameters). Similarly, copying makes object life-time management easier at the language-level, since destruction of each copy is determined by its location in the program, which is essential in C++ where garbage-collection is not built into the language.

When objects are copied in C++, the type of the copy is the static type used to reference the original object, i.e. in this case the type of the parameter or return value. With inheritance, this may end up slicing the object when the dynamic type of the object differs from the type used in copying [3, Ch. 27], which causes problems with C++ exceptions. These problems are discussed later in this section.

Pointer and reference parameters are usually used to pass objects whose dynamic type may differ, circumventing the slicing problem. However, pointer and reference passing is difficult to implement in a system with no shared memory. For these reasons pointer and reference parameters are not allowed as parameters for RPC in most systems, and are not discussed further in this article.

The static typing of copying combined with templates (which also use compile-time static typing) makes it quite straightforward to implement libraries for marshalling parameters and return values into data messages, sending these messages to the receiver, and then creating and unmarshalling those parameters and return values from the message. Template-based serialisation libraries already exist for C++ [4]. However, when exception handling is added to the picture, the situation is different.

When an exception is thrown, a copy is made of the exception object, based on the static type used in the throw statement. This semantics means it is not possible to properly throw an exception whose real dynamic type is not the same as the static type used to throw it. As an example, if the parameter of a catch clause is "const E & e", it is not possible to reliably re-throw the exception with throw e. If the type of the original exception is a subclass of E, the throw statement throws a sliced copy. This is the main reason why re-throwing is allowed only using the special syntax “throw;”, which re-throws the original exception object without copying it.

When a copy of an exception object is propagated out of a function call, its type is not statically defined by the function’s interface. Even exception specifications only declare base classes of allowed exceptions, but the dynamic type of exception objects can be any derived class. This means that information about an exception object’s dynamic type is needed when it is sent to a different address-space. Similarly, the receiver of an exception object (the caller) has to be able to re-create the exception without static compile-time knowledge of its type. Finally, the re-created exception object has to be thrown, again without knowing its type at compile-time.

All this makes passing exception objects across address-spaces more difficult in C++ than passing parameters and return values. Especially it makes the use of static template metaprogramming more challenging. The following sections analyse the problem further. A template-based solution using automatically generated dynamic factories is then described.
As mentioned, propagating an exception back to the caller in a distributed environment requires copying the exception object between address spaces. This requirement does not cause incompatibilities with normal C++ semantics, since the language explicitly states that copying of exception objects may occur even in normal C++ [5, §15.1/3]. However, the exception propagation mechanism is the only place in C++ where the dynamic type of a compiler-generated copy (a thrown exception) may differ from the static type used to destroy the object (in the exception handler).

Figure 1 shows a typical call sequence that ends in an exception. When a call request is sent to a server, its invoker code is responsible for interpreting the request data and then calling an appropriate C++ function that actually executes the requested service. If an exception is thrown within the function, it is the responsibility of the invoker to catch it and send it to the calling client. This requirement means that theinvoker has to be able to catch and handle different types of exceptions. In the figure, notes marked with an asterisk (*) denote places where knowing the dynamic type of the exception object is required.

The normal way to catch all exceptions in C++ is to use the `catch (...)` syntax. However, this mechanism is of no use here, because it does not give the error handling code any way to access the thrown exception object. Even the type of the thrown exception is not known to the exception handler.

The only other way for the function invoker code to catch exceptions is to require that all such exceptions are derived from a common base class (this is a usual requirement in many other object-oriented programming languages). The solution described in this article requires that exceptions thrown across address-spaces must be derived from a base class `ExceptionBase`, which allows derivation of mechanisms for serialisation, dynamic creation of exception objects, etc. Derivation from this class happens through an intermediate base class for template metaprogramming reasons and is explained later.

After the server has caught an exception using the common base class, it must be able to marshal it into a data stream and send this stream back to the caller as part of the return value message. Marshalling is implemented with a pure virtual function `marshal` in the exception base class. The invoker calls this member function and passes as a parameter the data stream, which is used to send the message.

The structure of RPC exception classes is shown as a UML class diagram in Figure 2. In addition to a common base class `ExceptionBase`, an intermediate exception base class template `Exception<...>` is provided. The programmer is expected to derive all RPC exceptions from this template, and give the actual exception class as a template parameter to the base class template (as indicated with the `<bind>` stereotype in the diagram), e.g.:

```cpp
class MyException : public Exception<MyException>
{ /* Normal exception class definition */
};
```

This idiom is often called the Curiously Recurring Template Pattern or CRTP [6], [7, §16.3]. Use of CRTP means that each derived exception class has its own unique instance of the base class template, and this base class instance knows statically the type of the derived class it belongs to.

The instance of the base class template knows statically the real type of the exception objects through the template parameter. Therefore it can implement the necessary virtual functions of `ExceptionBase` (like `marshal`), releasing the programmer of `MyException` from that duty.
CLIENT SIDE — UNMARSHALLING AND RE-THROWING

C++ is a statically typed language with very limited run-time reflective capabilities. This limitation means that when an object is created (including throwing an exception), the type of the object has to be known at compile time [5, §15.1/3].

However, creation of an RPC exception object on the client side has to be performed based on the dynamic type stored in the received message, which is inherently a run-time issue. Therefore, appropriate mechanisms for limited run-time reflection have to be built. The client-side mechanisms are more complex than those on the server side, where the template mechanisms just make it easier to write exception classes.
Unmarshalling exception objects

C++ has basic RTTI (Run-Time Type Information) constructs `dynamic_cast` and `typeid` [5, §5.2.7&8], but these cannot be used to create objects, only to ask about the dynamic type of an object. Therefore, the client side mechanisms for creating exception objects from received data must be written from scratch.

The solution is based on the Abstract Factory design pattern [8, pp. 87–95], which is automated using the template-based inheritance described earlier. The exception base class `ExceptionBase` keeps a static data member `registry`, a data structure that maps the type id of each exception class to an appropriate creation function, which in turn is able to create an object of the correct type. The mapping between exception classes and type ids is discussed later in section Implementation Issues. That section also analyses concurrency issues concerning the registry.

The base class has also a static member function `create`, which is used by the rest of the library to dynamically create and unmarshal exception objects when needed. When the client side receives a message containing an exception, `create` chooses the correct creation function from the registry based on the type id stored in the message. The chosen creation function then creates an exception object of appropriate type and unmarshals the object using the rest of the message data. If the message contains an exception object that is not found in the registry, the system returns a generic exception `UnknownRpcException` (this possibility is discussed in more detail in the next section).

Automating the creation of polymorphic factory

Normally, the use of a polymorphic factory would require exception classes to provide their own creation functions. [9, §1.17.10] However, the CRTP-based template inheritance described earlier can also be used to automatically create a creation function for each exception class. This approach makes
it possible to embed necessary mechanisms into an otherwise normal exception class, which can be seen from Figure 2.

Actual creation functions are located in the CRTP class template `Exception`. It has a private static function `createFunc`, which creates an exception object whose type is that of the `template parameter`, i.e. the actual exception object. Unmarshalling of data by the client is implemented using the default constructor and unmarshalling operator `>>`. For simplicity, the function assumes that creating and unmarshalling an exception object never fails (it would of course be possible to return a different exception in such a case).

```cpp
template <typename E>
ExceptionBase* Exception<E>::createFunc(IMsg& s) {
    E* e = new E; s >> *e; // Unmarshal message data into the object
    return e;
}
```

Using the `Exception` template makes it possible to automate the whole exception object creation and unmarshalling process. There is one `Exception` instance for each actual exception class, this instance contains a creation function needed to create such objects, and the creation function knows the static type of the object to be created.

The system has to make sure that each `Exception` instance registers its creation function to the base class `ExceptionBase`. The simplest possibility would be to list all creation functions in a compile-time list, which could be used to initialise the polymorphic factory. However, this approach would not be practical, since it would introduce a place in the program where knowledge on all RPC exceptions would have to be gathered. Since exceptions are often defined on module or class basis, having to update a separate list of RPC exceptions would be tedious and prone to errors. An alternative solution would be to initialise the polymorphic factory during program startup, and let each RPC exception class (or rather its creation function) be registered separately. This approach still requires manual registration, leaving the system vulnerable to missing registrations.

Fortunately the registration of each RPC exception class can be automated using templates. The system has to make sure that each `Exception` instance registers its creation function to the base class `ExceptionBase`. The base class provides a static protected function `addToRegistry`, which is used to add all creation functions to `registry`. The registration is handled using an intermediate class template `Registrator` nested inside the `Exception` template. This class template is otherwise empty, but has a default constructor to register the creation function of the enclosing class.

Each `Exception` template instance has a `Registrator` object as a static data member. When this data member is created during program startup, its constructor is executed and registers the `Exception` instance. However, the C++ template instantiation mechanism is partly based on lazy evaluation, and the consequences of this are discussed in the next section.

By using registrators, the system automatically registers all exception classes declared anywhere in the code. The only requirement is that the code of the client has to contain declarations for all exception classes the server might throw (this is of course the case if both the server and the client run the same executable). This includes declarations of thrown derived exceptions that the client catches by a base class reference. If the requirement is violated and the server throws an exception which is not declared in the client, that exception is replaced by a generic exception object `UnknownRpcException`.  

Copyright © ? John Wiley & Sons, Ltd.  
*Software Pract. Exper.* ?: ??:1–1
Template instantiation issues

In C++, static members of a class template are only instantiated if they are referenced in the program [5, §14.7.1]. This “laziness” is a useful feature, but in this case, the static Registrator data member is not referred to by any part of the program, since its sole purpose is to execute its constructor when the program starts. This means that registrators would normally never be instantiated at all.

The problem can be solved by utilising the fact that the type of a static member is always instantiated even if the member itself is not. The Exception class template declares an additional empty class template ForceInstance, which requires a pointer to a Registrator object as a non-type template parameter. A pointer to ForceInstance<&registrator> is then declared as a static data member of the Exception template:

```cpp
template <typename MyException>
class Exception
{
    static Registrator registrar;
    // Statically force instantiation of registrar
    template <Registrator*> class ForceInstance;
    static ForceInstance<&registrator>* notEverInstantiated;
};
```

The static data member notEverInstantiated is never referred to, so it is not instantiated and does not consume any memory or produce any code. However, instantiating its type in the class definition requires taking the address of the static data member registrar, which causes registrar to be instantiated.

This template metaprogramming mechanism makes the Exception class template act as a program generator [10], where just referring to it (instantiating it through inheritance) automatically generates necessary factory functions and registers them to the object factory.

Throwing the created exception object

When an exception is re-created on the client side using the polymorphic factory in ExceptionBase, an exception object is created and an ExceptionBase* base class pointer is returned. Now this object has to be thrown. Just like creating an object, throwing an exception in C++ requires static compile-time knowledge of its type [5, §15.1/3].

ExceptionBase declares a pure virtual function throwSelf. Its implementation in the Exception class template simply downcasts *this to the actual exception class type (the real dynamic type of the object) and then throws the exception object itself. Using these mechanisms, the client code can create a received exception object using create in ExceptionBase and throw the exception object by calling its throwSelf. Throwing the object copies it automatically, so the original exception object can be destroyed (preferably automatically during stack unwinding). Alternatively the created exception object can be stored for throwing it later. This kind of delayed throwing is necessary in an asynchronous environment using futures for delayed return value passing [11].
EXCEPTION HIERARCHIES

In this approach, a separate base class instance of `Exception` is used for each concrete exception class. Thus it is possible to generate all type-based exception handling code automatically during compile time. However, it also means that classes derived from the `Exception` class template are not useful as base classes themselves.

This restriction comes from the fact that the `Exception` instance has to get the type of the derived exception class as its template parameter. If further derivation would be allowed, this would not be true for classes derived from the original exception class. For this reason each RPC exception class has to be a leaf class in the inheritance hierarchy.

Exception hierarchies are extremely important in exception handling, so allowing them is important. The problem can be solved by using multiple inheritance. The structure of the solution is shown in Figure 3. The gray oval shows a “local” exception hierarchy, which is not aware of RPC issues. Actual RPC exception classes are then derived from both the local exception class and the `Exception` template instance. This is shown with classes `E1Rpc` and `E2Rpc`. Class `E3Rpc` is an example of a class which is known to be a leaf class, so it can be added directly to the local hierarchy and does not need an intermediate base class.

Using this kind of hierarchy is quite straightforward. All exception handlers can catch exceptions from the “local” exception hierarchy, so existing code does not necessarily have to be updated. Code whose exceptions cannot end up propagating to other address-spaces can also throw these exceptions. Throwing exception objects derived from `ExceptionBase` is only necessary in places where it is possible that an exception is sent to another address-space. These objects are instances of their base classes, so they can also be caught by exception handlers unaware of RPC issues:

```cpp
try {
    int result = activeObject.remoteCall(); // May throw
    catch (const E1L& e) { /* Also catches E1Rpc */ }
}
```

Generation of RPC exception classes from normal exception classes can be automated using templates. The class `E4L` and the template `RpcExcp<E4L>` show this in the class diagram. The template `RpcExcp` is otherwise empty, but it is derived both from its template parameter and the `Exception` template instance. The result is a mixin-like class, which implements the original exception class and contains necessary mechanisms for cross-RPC propagation:

```cpp
template <typename LocalExcp>
class RpcExcp : public LocalExcp,
    public Exception< RpcExcp<LocalExcp> >
{ /* Template constructors for parameter passing */ };
```

The `RpcExcp` template provides template constructors that simply call appropriate constructors of the local exception class. This way `RpcExcp<E>` can act as an RPC counterpart of an exception class `E`, and `RpcExcp<E1L>` could have been used directly in place of `E1Rpc`. Of course the programmer still has to write necessary marshalling and unmarshalling operators for the local exception class to make serialisation possible. The `RpcExcp` template is enough to create the rest of mechanisms for passing exceptions over RPC boundaries.
The `RpcExcp` template can also be used to catch original exceptions and re-throw them in RPC form. The following works if the type of the catch clause parameter is the same as the real dynamic type of the thrown exception object. Otherwise copying the object ends up slicing it:

```cpp
try { /* Code that throws an exception from "local" hierarchy */ } catch (const E4L& obj) { throw RpcExcp<E4L>(obj); }
```
PERFORMANCE MEASUREMENTS

Performance of the solution was tested by repeatedly calling a simple function that simply returned an integer or threw an exception depending on the test. Measurements were made for three test cases. The first test was performed with no exception handling code and with the compiler’s exception support disabled. The second test was performed with necessary exception handling code included but without actually throwing any exceptions. In the third test an exception was thrown from the function each time.

Since exception handling in normal C++ is often several orders of magnitude slower than normal return, exceptions are not usually used in time-critical segments of code. Therefore, it is not reasonable to compare exception handling performance directly to cases where exceptions do not occur. However, measuring the normal return gives an estimate for the normal function call overhead.

Each of these three tests were run in three different setups. The first setup used normal C++ function calls and exception handling. The second setup was a modified program that used serialisation for parameter and return value passing, and the mechanism described in this article for exception propagation, but within the same address space. The third setup was to use KC++ concurrent C++ system [1], for a real RPC call between address spaces. KC++ uses active objects executing concurrently in separate address spaces, and futures as an asynchronous communication mechanism. However, only synchronous calls were used in this test to make results easier to compare with each other (for asynchronous exception handling in KC++, see [11]). The KC++ tests give a realistic estimate for RPC-type overhead in remote invocation and having to pass serialised data to another address-space. The source code for the current test version of KC++ can be obtained from the author, if needed.

All tests were run under Linux Fedora Core 3 with AMD Athlon XP 2100+ processor and GCC 3.4.3 compiler with -O3 optimisation. The code calling the test function and the test function itself were placed in separate compilation units in order to make sure that the compiler could not use inlining to optimise away the actual function call. Test results were interpreted using the K-best method [12, Ch. 9] — each test program was run $N$ times and if the relative difference of $K$ best results was less than $\epsilon$, the result was accepted as “representative”. Values used for the test were $N = 20$, $K = 3$ and $\epsilon = 0.01$. The results are shown in Table I. Results in the table were rounded to two significant digits.

Tests 1, 4, and 7 were performed with the compiler’s exception handling support turned off. They are meant to represent the base line on top of which exception handling overhead is added. Tests 2, 5, and 8 were performed with the compiler’s exception handling support on, but without throwing exceptions. Finally tests 3, 6, and 9 show the case where the function returned by throwing an exception instead of returning a value.

Results 1–2 and 4–5 show that as many compiler writers claim, exception handling (try-catch blocks with exceptions enabled in the compiler) does not affect performance as long as exceptions are not actually thrown. (Note: Times for tests 4 and 5 are practically the same, but rounding to two significant digits caused a small difference in the time per call figure.)

Results 1–3 show that throwing and catching an exception in this test is four orders of magnitude slower than passing a normal return value. This result is similar to those mentioned elsewhere: “Throwing an exception may be as much as three orders of magnitude slower” [13, Item 15]. The situation is even more dramatic in this test. A probable cause for this is that the function is extremely simple and returns just an integer. This means that function call overhead is very low when a return
Table I. Results of performance tests

<table>
<thead>
<tr>
<th></th>
<th>Test</th>
<th># of rounds</th>
<th>Time (s)</th>
<th>Time/call (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>No exception handling</td>
<td>$2 \times 10^{10}$</td>
<td>23.23</td>
<td>1.2</td>
</tr>
<tr>
<td>2.</td>
<td>Excp handling, no throw</td>
<td>$2 \times 10^{10}$</td>
<td>23.23</td>
<td>1.2</td>
</tr>
<tr>
<td>3.</td>
<td>Excp handling and throw</td>
<td>$2 \times 10^{6}$</td>
<td>23.14</td>
<td>12000</td>
</tr>
<tr>
<td>4.</td>
<td>Serialisation, no excp handling</td>
<td>$2 \times 10^7$</td>
<td>19.67</td>
<td>980</td>
</tr>
<tr>
<td>5.</td>
<td>Serialisation, no throw</td>
<td>$2 \times 10^7$</td>
<td>19.70</td>
<td>990</td>
</tr>
<tr>
<td>6.</td>
<td>Serialisation and throw</td>
<td>$4 \times 10^7$</td>
<td>14.97</td>
<td>37000</td>
</tr>
<tr>
<td>7.</td>
<td>KC++, no exception handling</td>
<td>$1.2 \times 10^6$</td>
<td>20.23</td>
<td>17000</td>
</tr>
<tr>
<td>8.</td>
<td>KC++, excp handling, no throw</td>
<td>$1.2 \times 10^6$</td>
<td>20.04</td>
<td>17000</td>
</tr>
<tr>
<td>9.</td>
<td>KC++, excp handling and throw</td>
<td>$2.5 \times 10^7$</td>
<td>14.93</td>
<td>60000</td>
</tr>
</tbody>
</table>

value is used. Exception handling in modern C++ compilers does not cause performance penalties when exceptions are not thrown, but this results in a heavy overhead when an exception is thrown: “All associated run-time costs occur only when an exception is thrown. However, because of the need to examine potentially large and/or complex state tables, the time it takes to respond to an exception may be large, variable, and dependent on program size and complexity.” [14].

Result 6 shows that using the exception serialisation mechanism described in this article causes exception handling to perform 3 times slower than with normal exception handling. This increase is as expected, since serialisation inevitably means that the exception must be thrown twice, once in the test function and another time in the caller after unmarshalling. The remaining overhead is explained by serialisation and dynamic creation of the exception object.

Comparing result 6 to results 4–5, which show the overhead of serialisation, the performance cost of an exception is quite acceptable, since it is only 37 times slower than normal serialised return, compared to the 10000 times slower with no serialisation.

Results 7–9 show performance tests implemented using the exception mechanism in KC++. The test program consisted of two (active) objects running in different processes and thus different addressspaces. One object repeatedly called a method of the other object in synchronous RPC fashion. The method either returned a normal return value (7–8) or threw an exception (9).

KC++ uses POSIX message queues for communication between processes, including method calls, their parameters, return values, and exceptions (the latter two through futures). Since the caller and callee execute in separate processes, even synchronous calls include context switching in the operating system. This makes the measurements fluctuate much more than in the single-threaded experiments. Therefore the results 7–9 should not be interpreted as precise timing values. However, they show that adding exception support to the system causes no substantial performance penalty when exceptions are not thrown (tests 7–8).

Propagating an exception in KC++ takes 3.6 times longer than returning a value. Compared to the earlier 37 times difference this may seem to be low, but it is explained by the much greater overhead caused by message queues and process dispatch. The additional overhead of exception handling is 43
ms for KC++ and 36 ms for the simple serialisation test, so they are close to each other. This means that
the mechanism described in this article adds little overhead to exception handling in an RPC system,
relative to the cost of communication.

IMPLEMENTATION ISSUES

This section discusses implementation details concerning the exception class registry needed for re-
creation of exception objects on the client side. It addresses the issues of mapping between types and
type id strings, as well as concurrency.

The exception registry is used to re-create exception objects from marshalled data, which requires
each exception class to be associated with a unique type id that identifies the type of the marshalled
exception object. In this respect, the situation is identical to other serialisation mechanisms. C++
and its RTTI provide a method to create a string representing the “name” of a given type,
namely typeid(Type).name(). Unfortunately the C++ standard defines this string as “an
implementation-defined NTBS” (Null-Terminated Byte String) [5, 18.5.1]. The standard itself does
not guarantee that this string is unique or even that it stays the same if the code is compiled again.

In practice, the situation is much better than what the standard requires. In most current C++
compilers, the type id strings are unique. This is easy for the compiler to implement, since it has
to produce unique type ids for the linker anyway. In some compilers, uniqueness is not guaranteed for
nameless classes or local classes (classes with no external linkage), but such classes are hardly useful
as exception classes.

C++ ABIs (Application Binary Interface) are designed to promote compiler interoperability, and the
format of the type id string is included in many ABIs. For example, the Itanium C++ ABI [15] used by
many compilers specifies that type mangling used in object files is also used for type id strings. With
compilers implementing such a common ABI, the type id strings stay constant between files compiled
with different compilers.

However, in some cases it would be best not to rely on compiler-generated type ids. This is obviously
the case if the compiler implementation does not guarantee unique type id strings. In addition to this, if
the exception mechanism in this article is used together with a serialisation library using user-provided
type ids, having user-defined exception ids would be consistent.† For these reasons the base template
Exception accepts a user-defined type id string as an optional parameter. If given, the registry uses
the parameter to identify the exception class. If the parameter is omitted, compiler-generated C++ type
id string is used. It would be possible to use compile-time selection to make the type id parameter
optional only in compilers that are known to provide unique type id strings.

An example of a user-defined type id string is shown below (the value for MyExcp_typeid would
be defined in the same place where the code of MyExcp is located):

```
extern const char MyExcp_typeid[];
class MyExcp : public Exception<MyExcp, MyExcp_typeid> // ...
```

†Serialisation libraries aimed for persistence usually require user-provided type ids, because type ids should remain the same
between different versions of the program. The same can also be true in distributed systems where subsystems run different
versions of the software.
In a concurrent environment, possible synchronisation and mutual exclusion issues must be analysed. The registry is a static member of the exception base class, so it is generated automatically when the program starts. Similarly all derived exception classes register themselves to the registry before the main routine begins, using the registrator mechanism described earlier in this article. After initialisation the registry remains constant, so in a distributed environment registries remain in a consistent state.

If processes are started as separate programs, each program has its own registry. If processes are created later using fork or similar mechanisms, all processes automatically get a copy of the already initialised registry. Even a shared-memory environment would not have synchronisation problems, though the mechanism described here is not intended for such an environment. The contents of the registry remain static during program’s execution, so sharing the data structures would not cause problems (depending of course on the thread safety properties of the compiler and its libraries).

Some C++ environments allow dynamic linking of libraries into an already running program. In such environments, the registry mechanism could be enhanced to allow the client to dynamically link in new exception classes, when they are first encountered. However, in a shared-memory environment this would require additional mutual exclusion to atomically update the registry.

RELATED WORK

Several serialisation implementations exist for C++ and other languages. Many of these libraries are clearly written with object persistence in mind. This means that they are designed for serialisation and restoration of values, which is common for both persistence and RPC. However, this means that these libraries do not specifically handle propagation of exceptions. For example, many serialisation libraries are completely separate from method/function invocation mechanisms.

The idea to provide a virtual throwSelf function in order to get around static typing in the throw statement is very logical. Therefore it is no surprise that it is used in many places, e.g., it is part of the CORBA C++ Language Mapping Specification [9, §11.9]. Herb Sutter and Jim Hyslop also describe the solution in their C/C++ User Journal column [16]. The idea to automate the throwSelf using CRTP inheritance is believed to be original.

The implementation of the exception factory described in this article resembles the implementation of Alexandrescu’s object factories in [17]. However, Alexandrescu’s factories are still based on user-provided creation functions. The idea to use the constructor of a static data member for registration of factories is loosely based on James Coplien’s exemplars [18]. However, the mechanism of this article automates the factories using inheritance and template metaprogramming.

While many object-oriented programming languages like C++ and Java support exception hierarchies through inheritance, CORBA does not support hierarchical exceptions in its IDL interface specifications. All exceptions are divided into user exceptions and system exceptions, and CORBA C++ mapping defines that all user exceptions are directly inherited from a single base class, making the hierarchy only one level deep. [19, §3.12] [9, §1.19]

The serialisation used in CORBA relies on user-written factory functions for creating objects during unmarshalling. For user-defined classes and types, it is the responsibility of the application programmer to write such factory classes, instantiate a factory from each, and then register those factories with CORBA run-time [9, §1.17.10].

Copyright © 200? John Wiley & Sons, Ltd.
Separate serialisation libraries exist for C++, such as Boost Serialization library [20], which contains support for versioned serialisation and restoration. The Boost Serialization library concentrates on serialisation of arbitrary values, so exceptions are not specially addressed in that library. It relies on several explicit mechanisms for dynamic factory creation, but does not require a common base class for serialised objects.

Similar serialisation of values using explicit dynamic factories is provided by the GNU Common C++ and its TypeManager class [21]. Yet another serialisation library is s11n [2]. It uses the same virtual function based (un)marshalling mechanism as the solution in this article, but relies on explicit creation of dynamic factories.

Although examples in this article use only synchronous RPC, the same mechanisms could be used for asynchronous calls also. Asynchronous exception handling causes additional problems, however. More information on these issues can be found in [11].

CONCLUSION

This article shows that exception propagation in RPC and similar calls with no shared memory can be implemented as a template-based library, including serialisation and dynamic creation of exception objects. The solution requires minimal additional code from application programmers and allows the use of existing exception hierarchies.

The presented solution is light-weight and implemented completely using C++ and its template metaprogramming mechanisms. Necessary object factories and virtual functions are generated automatically, which means no pre-processors, code generators or separate IDL specifications are needed. (Manual type id strings are needed in the source, if portability is needed for compilers whose ABI does not guarantee uniform format of C++’s typeid strings.)

Performance tests show that the efficiency of the solution is good in situations where exceptions are an acceptable mechanism. An RPC exception propagation takes about 40 times longer than normal RPC return, which is acceptable, especially considering that normal exception handling is 3-4 degrees of magnitude slower than normal return from a function.

The solution presented in this article is used in the KC++ concurrent active-object based system.

REFERENCES


15. C++ ABI Summary (Itanium C++ ABI)


20. Ramey R. Serialization (the Boost serialization library).

21. GNU Common C++ resources.