



Welcome to Connections!

ZeroC's Chief Scientist, Michi Henning, wrote a wonderful article for this second issue of *Connections*. In "To Slice or Not to Slice..." Michi first explores why some developers try to avoid interface definition languages, even though they form the

very foundation of distributed computing. Next, he analyzes the web services approach to interface definitions, and exposes them as the exact opposite of what many believe they are, namely that, in contrast to Slice or CORBA IDL, they are designed to be human-unreadable. For those of you who keep an open mind and are not yet too indoctrinated by the current XML craze, this article is a must-read!

As always, this issue of *Connections* also has interesting articles about Ice programming techniques. We expand on the first issue's chat server example, an application that we will refine further in upcoming issues. We also start a new series about thread safety, one of the most difficult programming disciplines. Stay tuned for more articles about this important subject in future issues of *Connections*.

As we expand Ice into new domains, we will use *Connections* to inform our user community about the latest developments at ZeroC. This issue explores how "IceGrid" will simplify the use of Ice in grid computing environments, while a future article will introduce you to "Ice-E," our solution for embedded systems with platform support for everything from cell phones to real-time operating systems.

As always, we would love to hear your opinion of our newsletter. Join our user forum at <http://www.zeroc.com/vbulletin/>, and let us know what you liked or disliked, and what you would like to see in future issues of *Connections*.

Marc Laukien
President
ZeroC, Inc.

Issue Features

To Slice or Not to Slice

Interface definition languages are a cornerstone of middleware: they define type systems and allow code to be generated that otherwise would have to be laboriously written by hand. Yet, some people seem to think that interface definition languages are not necessary and that web services can somehow make do without them. This article takes a closer look at the truth or untruth of this idea and contrasts the approaches taken by Ice and web services to define type systems.

Advanced Use of Glacier2

The second in a continuing series of articles on application development with Ice, this article addresses several limitations in the initial chat server implementation and demonstrates how to use Glacier2's advanced capabilities to improve the robustness of an Ice server.

Thread-Safe Marshaling

Making your Ice server thread-safe may require more than simply adding synchronization to your servant. This article describes the interaction between the Ice runtime and a servant and explains how to avoid a subtle thread-safety issue in your application.

Ice and Grid Computing

As the practical applications of grid computing continue to grow, ZeroC is introducing a new product that aims to simplify the deployment of Ice servers in grid environments. This article describes the features we expect to include in the initial release.

Contents

To Slice or Not to Slice...	2
Advanced Use of Glacier2	6
Thread-Safe Marshaling	11
Ice and Grid Computing	14
FAQ Corner	16

To Slice or Not to Slice...

Michi Henning, Chief Scientist

I recently had an e-mail conversation with a developer at a company that was considering Ice for a project. As is often the case, the company was also evaluating web services. During our conversation, the developer expressed concern that Ice might be rejected because his “management has gone cold on interface definition languages.”

I was quite puzzled by this remark—how can someone go “cold” on interface definition languages? By implication, this both means that interfaces do not need describing and that web services do not use an interface definition language, neither of which is correct. So, I thought it might be interesting to describe the purpose of interface definition languages, discuss the consequences of not using such a language, and to contrast Slice and WSDL.

Purpose of Interface Definition Languages

Type System Definition

Interface definition languages, such as Slice, WSDL, CORBA IDL, and DCE IDL, have a number of purposes. Depending on the middleware platform, these purposes may vary; but, irrespective of the middleware, *all* interface definition languages share one common purpose, namely, to define a *type system*. This means that the language describes the types of values that will be exchanged between client and server, as well as the operations that a client can invoke, and the types of values that are passed between client and server with each operation. Depending on the exact language, it may also provide mechanisms to describe type substitutability (inheritance) and support objects and polymorphism. (Slice and CORBA IDL do this, whereas DCE IDL and WSDL do not.)

The point of defining a type system is to ensure that client and server agree on the kinds of messages that are exchanged. Without a type system, communication would be impossible. For example, we may have an operation that expects to be passed an integer and a string. It is the type system that embodies this knowledge, and client and server must agree on it—if the server expects an integer and a string, but the client sends two doubles, things are simply not going to work.

Apart from defining a type system, interface definition languages can serve a number of other purposes.

API Definition

Interface definition languages can serve to define APIs: a compiler processes a definition and generates source code for a particular target language, such as C++. The generated code provides an API that the application code calls in order to manipulate values (such

as inserting a value into a sequence), invoke operations, manage memory, handle exceptions, and so on.

The rules that govern exactly how an interface definition results in a particular API are known as *language mappings*. Slice, CORBA IDL, and DCE IDL all define mappings for one or more languages, each using its own set of rules. (The quality and ease-of-use of the generated API varies considerably, but that is the topic of another article.) Note that the actual language mapping typically is *not* specified by the interface definition language itself, but is governed by a separate set of rules written down elsewhere (although, in the case of Slice, you can influence a few specifics of the mapping with metadata directives).

WSDL does not define language mappings—the specification is silent about how an application interacts at the programming-language level with a web service that is defined by WSDL. (The JAX RPC specification defines a mapping between WSDL and Java. However, the specification is not supported by many web services toolkits, and there are no standardized mappings for other languages, so each toolkit creates its own proprietary API.)

Protocol Definition

Interface definition languages can specify *protocols*. A protocol is a set of rules that determine the choreography of on-the-wire messages that are exchanged between client and server. For example, the Ice protocol specifies that each message begins with a message header and defines the contents of that header. Similarly, for each message type, the protocol defines a header that is specific to the type of message and that follows the message header. The protocol also defines rules for how messages are to be exchanged. For example, a reply message must be returned in response to a request message, and a connection opened by a client must be acknowledged by the server in the form of a validate-connection message before the client can send anything else. In other words, a protocol defines the layout and meaning of headers as well as a state machine that defines how and in what order messages must be exchanged.

Ice, CORBA, and DCE do not specify protocols as part of their respective IDLs. Instead, the protocol is determined by information that is implied or stored elsewhere, such as in configuration information for the run time. With WSDL, the protocol is specified as part of the `binding` element.

Serialization and Encoding Definition

The term *serialization* refers to how complex data structures, such as sequences and dictionaries, are to be transmitted between client and server. The emphasis here is on how individual data items are ordered. For example, a typical serialization rule would be “sequences are sent in increasing element order.” An alternative serialization rule would be “sequence elements can be sent in any order; each sequence element is preceded by its index.”

The term *encoding* refers to how individual data items are represented as bytes on the wire. For example, typical encoding rules would be “strings are sent in left-to-right order, with each character encoded in UTF-8 format” and “doubles are sent in little-endian byte order, encoded in IEEE format.”

Serialization and encoding are often jointly referred to as *marshaling* (with the inverse process known as *unmarshaling*).

As for protocols, Ice, CORBA, and DCE do not specify marshaling as part of their respective IDLs, whereas WSDL specifies the format of messages in its `binding` element.

Transport Definition

Underneath the message, protocol, and marshaling layers, we have a *transport* that is used for communication between client and server. The distinction between “protocol” and “transport” is often blurred—whether something is a protocol or a transport can depend on the perspective of the developer. (For example, to a middleware developer, TCP/IP is a transport, whereas, to a TCP/IP developer, Ethernet is a transport.) As far as middleware is concerned, any medium outside the middleware itself is usually viewed as a transport. From that perspective, TCP/IP, SSL, and UDP are all transports.

Ice, CORBA, and DCE do not specify transports as part of their respective IDLs, whereas WSDL specifies the transport in its `binding` element.

Enforcement of the Client–Server Contract

The prime motivation for the use of interface definition languages is to automate the chores of adhering to a protocol, serializing and encoding data, selecting an appropriate transport, and so on. Given an interface definition, a compiler can generate code that links with client- and server-application code to handle all of these chores. (The generated client- and server-side code is called *stubs* and *skeletons* or *proxies* and *stubs*, respectively, depending on the middleware.)

Generated code reduces development cost because generated code does not need to be written by developers. Moreover, interface definition languages raise the level of abstraction because (exempting WSDL) they shield the developer from low-level detail, such as choosing an appropriate data encoding. However, the biggest payback from interface definition languages is that generated code enforces the client–server contract *at compile time*: the generated stubs and skeletons (as a rule) have an API that is statically type-safe, meaning that any attempt to pass incorrect data between client and server (such as a double instead of a string) is caught when the program is compiled.

It is well documented that, over the life cycle of an application, the earlier an error in a program is detected, the lower the cost of fixing the error, so catching errors at compile time is valuable.

Can We Live Without Interface Definition Languages?

As we established earlier, it is essential that client and server agree on how information is to be exchanged between them. Seeing that the one purpose that is common to all IDLs is to define a type system and to generate code, the question boils down to whether we can live without a type system definition language and, consequently, without generated code.

The answer is “yes”: it *is* possible to write distributed applications without any kind of IDL. For example, Ice and CORBA both offer *dynamic invocation and dispatch* interfaces. These interfaces permit a developer to invoke operations from a client and to implement them in a server without ever writing a single line of interface definition. However, using these interfaces extracts a price: in essence, they require the developer to explicitly implement much of the functionality that is otherwise provided by generated code. The developer simply has to *know* that a particular operation expects an integer and a string (in that order), and has to write explicit statements that place the integer and the string on the wire on the client side and retrieve them again on the server side.

The problem with dynamic invocation and dispatch is that it rapidly becomes complex. It is easy for developers to make mistakes, such as sending parameters in the wrong order or of the wrong type, or to invoke an operation on an object that does not support that operation. Moreover, the definition of the types, operations, and parameters are no longer explicitly manifest in a convenient interface definition. To find out how the type system works, I have to read the client and server code and reconstruct the type system from that. Obviously, this is a lot harder and error-prone itself.

What makes it worse is that any errors are no longer detectable at compile time. Instead, error detection is deferred until run time, and errors are detected only if there is a test case to expose them. This means that, quite often, errors are not detected until well after an application is deployed, with sometimes disastrous cost blow-outs.

Still, there are applications that truly require dynamic invocation and dispatch, such as message routers, protocol bridges, firewalls, and publish-subscribe event services. In order to stay generic, such services cannot depend on a fixed type system that is defined prior to deployment because the type system may change at any time. However, these applications are in the minority. The overwhelming majority of applications operate on a fixed type system that is known in advance and does not change during the applications’ lifetime.

Given the cost and potential dangers of dynamic invocation and dispatch, you should consider its use very carefully. You should make the trade-off between error detection at compile time versus error detection at run time only if there is a significant payback in terms of flexibility, and only if that flexibility truly is required by your application.

TO SLICE OR NOT TO SLICE...

But back to the question of whether we can live without interface definition languages... Yes, we can live without interface definition *languages*, but we cannot live without interface *definitions*! Even when using dynamic invocation and dispatch, there must still be agreement between client and server about the type system. If we do not use an interface definition language, we must use some other way to enforce the agreement. For example, we can write a data model on a whiteboard and tell all developers to look at the board religiously (and some of them might even do that). Or we can just rely on programmers' good will (and good debugging skills). One way or another, the agreement must be enforced or the application will break.

The preceding discussion should make it clear that the phrase "going cold on interface definition languages" is strange indeed. In fact, even if we *do* use dynamic invocation and dispatch and forgo compiled stubs and skeletons, we would *still* be better off writing an interface definition and compiling it, even if we were to throw the generated code away. After all, that way, we could at least be sure that the definition is internally consistent (because it compiles), and we would have a definitive document containing the definition, written in a well-defined formal language that is easy to read.

Slice versus WSDL

Given that we have established the need for interface definitions (if not the need for interface definition languages), it seems appropriate to compare Slice and WSDL in the context of deciding which technology you should use. WSDL in the web services world serves the same need as IDLs for other middleware: it defines a type system and, thereby, enables code generation. (And, even with web services, very few people seriously suggest to not use WSDL.)

Orthogonality

Apart from defining a type system, WSDL also concerns itself with a number of other things. One of these is the `binding` element that defines protocol and marshaling rules for accessing a service. Another is the `port` element: a *port* is a definition of an endpoint at which a service can be reached, such as `<soap:address location="http://127.0.0.1:8988/myApp/myService"/>`. Yet another is the `message` attribute of a `portType` element: it defines whether a particular message is an ordinary twoway invocation, a oneway invocation, a twoway callback from server to client, or a oneway callback from server to client.

In contrast, "pure" IDLs, such as Slice, keep their focus on the type system and do not concern themselves with protocol, marshaling, endpoint, and invocation details. Which is better?

One of the golden rules of computing is *divide and conquer*: focus on one thing at a time, and solve orthogonal sets of problems using separate mechanisms. Such separation of concerns is valuable because it makes it possible to independently change things

in one problem set without affecting all the other problem sets. WSDL concerns itself with issues that are quite independent from each other, such as the type system, the protocol and transport, and the style of operation dispatch. If nothing else, this makes things more complex (because WSDL specifies more things than Slice), and it dilutes the focus of the developer: "What am I doing here? Am I specifying endpoints, or call dispatch, or types?"

A definite pragmatic downside of the WSDL approach is that a simple change can have significant impact on development time: changing a minor detail (such as the endpoint of a service) requires recompilation of the definition and, consequently, recompilation of all system components that depend on that definition. This often means that the majority of source files that make up an application require a rebuild, even though the specific change might not strictly require rebuilding most of these files. If you have sophisticated development tools that are intimately aware of the meaning of the various parts of a specification, this problem can be mitigated (but most development tools are not that smart).

Ergonomics

XML is often described as "human-readable" (a term that is frequently used as an advantage of SOAP, and generally used to justify the use of XML for all sorts of things). Consider the following [WSDL specification](#):

```
<?xml version="1.0"?>
<definitions name="StockQuote"
  targetNamespace=
    "http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/
    soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
  <schema targetNamespace=
    http://example.com/stockquote.xsd
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
      <complexType>
        <all>
          <element name="tickerSymbol"
            type="string"/>
        </all>
      </complexType>
    </element>
    <element name="TradePrice">
      <complexType>
        <all>
          <element name="price" type="float"/>
        </all>
      </complexType>
    </element>
  </schema>
</types>
<message name="GetLastTradePriceInput">
  <part name="body" element=
```

TO SLICE OR NOT TO SLICE...

```
"xsd:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
  <part name="body" element="xsd:TradePrice"/>
</message>
<portType name="StockQuotePortType">
  <operation name="GetLastTradePrice">
    <input message="tns:GetLastTradePriceInput"/>
    <output message="tns:GetLastTradePriceOutput"
  />
</operation>
</portType>
<binding name="StockQuoteSoapBinding"
  type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport=
      "http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetLastTradePrice">
    <soap:operation
      soapAction=
        "http://example.com/GetLastTradePrice"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="StockQuoteService">
  <documentation>My first service</
documentation>
  <port name="StockQuotePort"
    binding="tns:StockQuoteBinding">
    <soap:address location=
      "http://example.com/stockquote"/>
  </port>
</service>
</definitions>
```

This is quite a lengthy specification. Here is the equivalent Slice definition:

```
// My first service
interface StockQuoteService
{
  float GetLastTracePrice(string tickerSymbol);
};
```

You may have your own opinion on which specification is easier to read and write. (Personally, it did not take me long to make up my mind...) Even without knowing any detail about WSDL, it is immediately obvious that writing WSDL by hand would be prohibitively slow, so it needs to be generated from some other description. You might expect that a common way of doing this would be to have a higher-level language, such as Slice, and to translate that language into WSDL. However, this is not the case: many toolkits instead generate WSDL from Java source that is embellished with comments that embed the additional information required by WSDL.

The problem with this approach is that it results in *two* equally useless definitions of the type system: the source code, from which it is hard to extract the type system definition because the code is cluttered with executable statements, and the WSDL, which, at even the most modest level of complexity, is essentially unreadable. And, of course, the approach does not get rid of the interface definition language on which “management has gone cold”: instead, it carefully hides the interface definitions in amongst a big pile of source code, so they are out of sight.

Differences in Type Systems

Slice and WSDL do not have identical built-in types. For example, WSDL provides built-in types for things such as date and time. Whether this is an advantage is questionable: most programming languages do not have equivalent built-in types so, by necessity, language mappings must be supplemented with library code that emulates the missing types. In turn, this makes language mappings more complex to learn and use.

WSDL also provides unsigned integers. Unfortunately, these cause major headaches with languages that do not have the concept of unsigned integers, such as Java. Unsigned integers can cause interoperability problems: it is anyone’s guess what happens when a C++ client sends an unsigned integer that exceeds the range of a signed integer to a Java server. (Programmers usually deal with this problem using the ostrich approach—at least until they eventually run out of sand...)

A much more serious issue with WSDL, however, is that it lacks any notion of object-orientation. The reason for this is historical: one of the driving forces behind web services was a wish to integrate functionality provided by web servers into client applications. Without a well-defined API to get at the functionality, developers were forced to parse HTML, which is inefficient as well as complex and error-prone: if a web server’s HTML pages are updated, the code easily breaks. Web services were meant to address this problem but, due to the designer’s mind-set and background, object-orientation was never given serious consideration. (Ironically, SOAP originally stood for *Simple Object Access Protocol*, but people kept pointing out that there are no objects in SOAP; the solution to this dilemma was to declare that “SOAP” simply stands for itself and is not an acronym.)

Proponents of web services maintain that there is no need for object-orientation, claiming that this has the advantage of simplicity. In effect, the only notion of “object” in web services is that of an entire service. However, true object-oriented features, namely inheritance, polymorphism, late binding, and type substitutability, are non-concepts in WSDL. In contrast, Slice supports these features, so you can create a fully object-oriented view of your application that seamlessly spans the client–server boundary, without having to map down to a procedural and data-centric view for remote communication.

TO SLICE OR NOT TO SLICE...

One of the aims of web services is to provide a global e-commerce infrastructure. By necessity, such an infrastructure, once finally established, will be extremely complex and represent one of the largest collections of inter-connected computer systems on Earth. In the light of decades of research and experience that have shown object-orientation to be a powerful abstraction mechanism, it seems questionable whether throwing it away was a wise decision. Certainly, one is reminded of the saying about the baby and the bath water...

Syntax

So why on earth would anyone use XML to describe type systems if it is so inconvenient? Partly the answer can be found in history: the development of web services rode the XML craze of the late 90's, and if something wasn't XML, it was automatically passé, regardless of whether XML was a suitable syntax or not. (SOAP is a case in point, but I digress...)

However, there is a legitimate case for having XML descriptions of type systems: XML is a universal syntax—a lingua franca that is useful for the exchange of information between various tools. This is useful (though not essential) when it comes to interfacing development environments with things such as UML generators or metadata repositories. In this respect, WSDL is better than Slice, albeit to the detriment of clarity, usability, and development and maintenance cost.

Summary

As we have seen, web services cannot reasonably live without interface definition languages, and neither can middleware platforms such as Ice or CORBA. The need for interface definition languages is in the nature of the beast, whether you love them or hate them. Fundamentally, WSDL and Slice are equivalent, in that they both define type systems and enable code generation, but WSDL also specifies many other details that are orthogonal to type systems and better kept elsewhere.

The human-unreadable nature of XML requires the use of additional tools for generating XML, either from source code or from another definition language that provides sufficient detail to capture all the information required by WSDL. Regardless, this obscures the actual interface definitions, either because they are buried in the source code, or because they are cluttered with additional information, such as endpoint details.

The lack of object-oriented features in WSDL is a serious limitation. In effect, web services revert back to the procedural and data-centric programming style of the seventies. Claiming such a retrograde step as an advantage seems self-serving in the extreme.

One advantage of WSDL over Slice is that, due to its standard syntax, it can be exchanged among various development tools. However, unless you have an overriding need to do this, the disadvantages of WSDL are likely to outweigh its one advantage.

At any rate, the notion of “going cold on interface definition languages” is nonsense—interface definition languages are a fact of life with distributed systems, and they will not go away, no matter how much management might wish them to. And, notwithstanding any personal bias on my part, I believe that you should be able to Slice your cake and eat it, too.

Advanced Use of Glacier2

Matthew Newhook, Senior Software Engineer

Introduction

In Issue 1 of *Connections*, we presented a very simple chat server that used Glacier2 to satisfy its session management requirements. This article describes how to make that server more robust and better suited for real-world deployments.

Firewalls

In a realistic deployment, the computer that hosts the chat server will typically reside in a private network and be protected by a firewall. Clients would interact with the server through a reserved port on the firewall that is forwarded to the host computer. However, this strategy requires that all proxies published by the server point at the firewall's address and port, and not at the inaccessible internal address of the server itself (see Figure 1).

With a Glacier2 deployment, you can accomplish this using a one-line addition to the Glacier2 configuration. If the firewall address is `firewall.mydomain.org` and port 13500 is forwarded, you would add the following property to the Glacier2 configuration:

```
// Glacier2 configuration
Glacier2.Client.PublishedEndpoints=ssl -h
firewall.mydomain.org -p 13500
```

A client-side firewall (if present) would not pose a problem for our application because the connection between the client and Glacier2 is bidirectional, as described in the previous article.

Transport Issues

We must address some transport issues to ensure that the chat server meets our reliability and performance requirements.

Assume for the moment that we don't use Glacier2 in our application, and instead the client connects directly to the server. Now consider the following hypothetical code:

```
// C++
void
ChatRoom::enter(const ChatCallbackPrx& callback)
{
    Lock sync(*this);
    _members.push_back(callback);
}

void
ChatRoom::message(const string& data) const
{
    Lock sync(*this);
    list<ChatCallbackPrx>::const_iterator p;
    for(p = _members.begin();
        p != _members.end();
        ++p)
    {
        try
        {
            (*p)->message(data);
        }
        catch(const LocalException&)
        {
        }
    }
}
```

The `ChatCallback::message` invocation in this scenario is a twoway invocation that locks the `ChatRoom` object for the duration of the call. Furthermore, since the request is sent over the Internet, we expect it to take some time. Given a latency of, say, 25ms, this implementation would be limited to 40 messages per second—not nearly enough. Even worse, since no timeouts are configured, a misbehaving client that has a network problem or maliciously blocks in the `message` operation may delay the response indefinitely! Clearly, this is not an acceptable solution.

What if we were to use a oneway proxy instead of a twoway proxy? Unfortunately, this would not fix the problem. A client that fails to read messages from the network would cause the protocol stack to fill up and, eventually, TCP/IP flow control would cause the server's call to `message` to block.

An alternative solution is to buffer message invocations using a separate server thread for each client. That way, if a client misbehaved during the message invocation, only the offending client would be affected. However, this is non-trivial to code and introduces scalability problems, such as running out of threads.

Figure 1: Glacier2 Deployment with Firewall

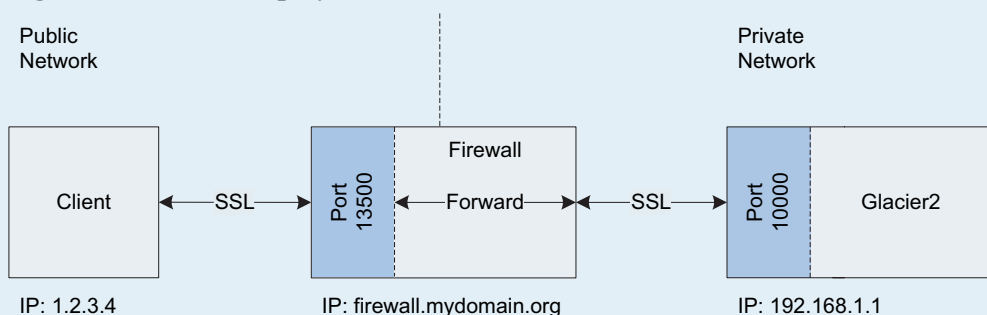
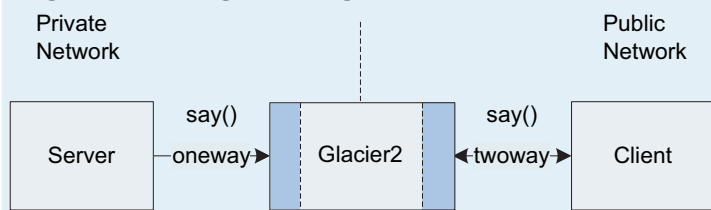


Figure 2: Message Routing with Glacier2



How Glacier2 Helps

Fortunately, Glacier2 can help you solve this problem. The server’s invocation of `message` is delivered in two steps: the server sends the invocation to the Glacier2 router, and then the router sends the invocation to the chat client. The chat server code uses the `ChatCallback` proxy as a oneway proxy instead of a twoway proxy, which means the server sends the data to the Glacier2 router in a oneway invocation. However, the Glacier2 router forwards the invocation to the client as a twoway invocation (Figure 2).

You may be wondering whether this strategy is safe. The answer is yes: this setup is safe in that a misbehaving client or a slow network connection cannot cause the entire server to block, as long as Glacier2 is using buffered mode. However, for reasons we describe later, we do not recommend this strategy.

In its buffered mode (which is the default), the Glacier2 router queues requests and creates a separate thread for each connected client that forwards the client’s requests. In its unbuffered mode, the router forwards each request immediately, which presents some limitations that are unacceptable for the chat server: if we were to use unbuffered mode and a client were to block for any reason, all other clients connected through the router in question would also stop receiving messages. This would occur regardless of whether the messages between the router and the client were twoway or oneway. Note that using oneway messages merely delays this problem, but does not prevent it (see the FAQ question “*Why can oneway requests block?*” in this issue for more information).

You can set the buffering mode independently for each direction that requests travel. By default, buffered mode is enabled for both directions, so using a oneway call to Glacier2 and a twoway call from Glacier2 to the client is safe in the default configuration. (See the [Ice manual](#) for more information on the buffering modes.)

Oneway Messages Are Not Reliable

Is using a oneway proxy to send messages to the Glacier2 router a good idea? If Active Connection Management (ACM) is used, oneway messages can be lost without any notification to the application. But, for scalability reasons, we need to use ACM on the back end, including between the Glacier2 router and the chat server. Therefore, we have to use twoway messages between the chat server and the Glacier2 router.

However, as previously discussed, using twoway messages in the entire `ChatCallback::message` call chain is a bad idea. Given this, the only other place we could use a oneway message is between the Glacier2 router and the client. However, can’t these messages be lost too? The answer is clearly yes. However, this situation is different because ACM is disabled between the Glacier2 router and the client, therefore messages can only be lost if the client terminates; losing a chat message in this situation is acceptable.

Since oneway messages are a complex topic, we have devoted the FAQ corner of this issue to them. You can find additional details there.

Recommended Configuration

Now to the heart of the matter: the recommended configuration for the chat server is to use a twoway message to the Glacier2 router. This provides maximum reliability and does not cause problems for the `message` implementation because Glacier2 is always well-behaved and does not block the server for any length of time. Ideally we should not hold the lock inside `message` for the duration of the RPC (to allow for better concurrency), but we will address this in a future article.

For the communication between Glacier2 and the chat clients, we’ll use oneway messages. This provides maximum throughput and avoids problems due to network latency or malicious clients. In addition, the Glacier2 router must use buffered mode for server-to-client calls.

How is the Glacier2 router told to make oneway invocations back to the clients? This is accomplished by supplying a request context containing a special key that is recognized by Glacier2. The key is “`_fwd`” (which stands for “forward”), and the value is “`o`” (which stands for “oneway”):

```

// C++
void
ChatRoom::message(const string& data) const
{
    Lock sync(*this);
    Context context;
    context["_fwd"] = "o";
    list<ChatCallbackPrx>::const_iterator p;
    for(p = _members.begin();
        p != _members.end();
        ++p)
    {
        try
        {
            (*p)->message(data, context);
        }
        catch(const LocalException&)
        {
        }
    }
}
  
```

If all calls on the proxy are to use this `Context`, then a simpler and

safer way to accomplish the same thing is to change the context on the proxy itself:

```
// C++
void
ChatRoom::enter(const ChatCallbackPrx& callback)
{
    Lock sync(*this);
    Context context;
    context["_fwd"] = "o";
    ChatCallbackPrx cb =
        ChatCallbackPrx::uncheckedCast(
            callback->ice_newContext(context));
    _members.push_back(cb);
}
```

Now any invocation on the callback proxy uses the context established with `ice_newContext` and there is no need to explicitly pass the context to each call of `message`.

Batch Oneway Messages

Glacier2 supports sending oneway messages in batches. In this mode, the router groups together several messages and periodically sends them in one batch. This can potentially send multiple chat messages in a single protocol message, which is more efficient and consumes less bandwidth. To enable batch oneway requests, change the “o” to an “O”.

```
// C++
void
ChatRoom::enter(const ChatCallbackPrx& callback)
{
    Lock sync(*this);
    Context context;
    context["_fwd"] = "O";
    ChatCallbackPrx cb =
        ChatCallbackPrx::uncheckedCast(
            callback->ice_newContext(context));
    _members.push_back(cb);
}
```

Note that batching of messages does not work unless you use Glacier2's buffered mode. You should also set `Glacier2.Server.SleepTime` so that messages get a chance to accumulate for batching purposes.

```
// Glacier2 Configuration
Glacier2.Server.SleepTime=500
```

Compressed Messages

Since the data we send through our chat clients is plain text, it tends to compress very well and therefore it's a good idea to enable protocol compression. You may be tempted to write the following, but beware, it is incorrect:

```
// C++
void
ChatRoom::enter(const ChatCallbackPrx& callback)
{
    Lock sync(*this);
    _members.push_back(
        ChatCallbackPrx::uncheckedCast(
            callback->ice_compress(true)));
}
```

This only enables data compression from the chat server to the Glacier2 router. However, since we assume the internal network to be extremely fast, this would actually slow things down because, on fast networks, it is quicker to send the uncompressed data.

Instead, we want to enable compression between Glacier2 and the chat clients. To do this, we must set another key in the request context. For compression the value is “z”, and we'll add this to the context information that is set on the proxy in `ChatRoom::enter`. (For efficiency reasons, the Ice protocol does not compress messages smaller than 100 bytes, even with this key set. However, since we are using Glacier2 in batched mode, messages will typically be larger and therefore will be compressed, especially in a busy chat room.)

```
// C++
void
ChatRoom::enter(const ChatCallbackPrx& callback)
{
    Lock sync(*this);
    Context context;
    context["_fwd"] = "Oz";
    ChatCallbackPrx cb =
        ChatCallbackPrx::uncheckedCast(
            callback->ice_newContext(context));
    _members.push_back(cb);
}
```

Now the information is forwarded by the Glacier2 router in compressed form in batched oneway invocations. The calling situation we have now is shown in Figure 3.

Denial Of Service

We saw earlier how Glacier2 can help provide protection against denial-of-service attacks when the server sends messages to clients. However, the attacks can occur in the opposite direction as well: a malicious client could flood the server with requests. Since it is not reasonable for a client to send more than one message or so per second, you should set the `Glacier2.Client.SleepTime` property.

```
// Glacier2 configuration
Glacier2.Client.SleepTime=500
```

This causes the sending thread in the Glacier2 router to sleep for 500ms after forwarding all queued requests from a client. Note that this property is only relevant when using buffered mode.

Note that the use of sleep timers does not introduce any additional latency unless the client tries to flood the server. When a request is received by the Glacier2 router, it is forwarded immediately, and then, as a safeguard against message flooding, the router sleeps for the configured sleep time. In other words, as soon as the user presses the return key, the message is sent and Glacier2 immediately forwards it to the chat server. A delay would occur only if the user sent another message within 500ms, which is not the case in normal conversation.

Object Access

Once a client has established a session, Glacier2's default behavior allows that client to access any objects hosted by any servers accessible from Glacier2. For example, if a chat user somehow obtained a proxy for another user's session, that user could pretend to be the other user by calling the proxy's `say` method.

As a security measure, Glacier2 can be configured to filter requests based on object identity. An identity is composed of two string members, `name` and `category`, but Glacier2's filtering mechanism considers only the `category` member. The primary configuration parameter is `Glacier2.AllowCategories`, which specifies a list of categories that all clients are permitted to use; invocations on objects having other categories are rejected. However, this capability isn't very useful for the chat server as it currently stands. What we need instead is to restrict a client's access to those objects that are in its current session, therefore we need a unique category for each client.

The solution is to set the category field to the user id and to add the user id to the list of allowed categories, which is accomplished by setting the configuration property `Glacier2.AddUserToAllowCategories`. The property has three possible values: 0, meaning disabled; 1, meaning add the user id to the list of allowed categories; and 2, meaning add the user id prepended with an underscore to the allowed categories.

Using a value of 2 is useful because it establishes a set of reserved categories for the server back end. For example, assume that we set `AddUserToAllowCategories` to 1. If the chat server had objects with a category value of `dummy`, it would be possible for a client to access those objects by creating a user with the id `dummy`. On the other hand, if we set the property to 2, we can ensure that

this situation cannot happen as long as we avoid using a leading underscore in the categories of any protected chat server objects.

To enable this behavior, you must modify the Glacier2 configuration as follows.

```
// Glacier2 configuration
Glacier2.AddUserToAllowCategories=2
```

You also need to create the session with the correct object identity: the category must be “_” followed by the identity of the user:

```
// C++
virtual Glacier2::SessionPrx
create(const string& userId,
       const Current& current)
{
    Identity id;
    id.category = "_" + userId;
    id.name = IceUtil::generateUUID();
    return Glacier2::SessionPrx::uncheckedCast(
        current.adapter->add(
            new ChatSessionI(userId, id));
}
```

Timeout Detection

As discussed in the previous article, Glacier2 can automatically destroy inactive sessions. The relevant Glacier2 configuration property is shown below:

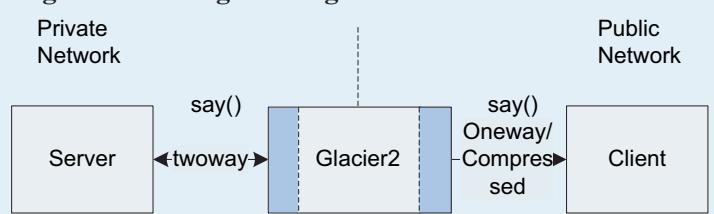
```
// Glacier2 configuration
Glacier2.SessionTimeout=30
```

This means that Glacier2 will destroy a session if it has no activity for more than 30 seconds. Of course, 30 seconds is much too low for a real deployment, but for testing purposes it is advantageous to have a short timeout. We can expect many users to be logged into several chat rooms and to be idle for long periods, so increasing the timeout period does not help. However, disabling session timeouts is not an option either, because network problems, bugs, and malicious clients could cause the server to fill up with stale sessions.

The solution is to ensure that the session gets used periodically while the client application is alive. We can do this by starting a dedicated thread in the client that calls `ice_ping` on the session at regular intervals. The interval should be less than the Glacier2 session timeout—in this case we pick 20 seconds to allow 10 seconds of breathing room for network latency.

```
// C++
class SessionPingThread :
    public IceUtil::Thread,
    public IceUtil::Monitor<IceUtil::Mutex>
{
public:
```

Figure 3: Message Settings



```
SessionPingThread(
    const Glacier2::SessionPrx& session) :
    _session(session),
    _timeout(IceUtil::Time::seconds(20)),
    _destroy(false)
{
}

virtual void
run()
{
    Lock sync(*this);
    while(!_destroy)
    {
        timedWait(_timeout);
        if(_destroy)
        {
            break;
        }
        try
        {
            _session->ice_ping();
        }
        catch(const Ice::Exception& ex)
        {
            break;
        }
    }
}

void
destroy()
{
    Lock sync(*this);
    _destroy = true;
    notify();
}

private:
    const SessionPrx _session;
    const IceUtil::Time _timeout;
    bool _destroy;
};

typedef IceUtil::Handle<SessionPingThread>
    SessionPingThreadPtr;
```

This thread is started after establishing the Glacier2 session:

```
// C++
Glacier2::SessionPrx session = ...;
SessionPingThreadPtr ping =
    new SessionPingThread(session);
ping->start();
```

We destroy the thread before `Application::run` terminates:

```
// C++
SessionPingThreadPtr ping = ...;
ping->destroy();
ping->getThreadControl().join();
```

Conclusion

By using the advanced features of Glacier2, you can protect your servers from denial-of-service attacks, ensure that your servers are highly scalable, and protect sensitive back-end objects without writing any additional code.

Stay tuned as the chat server continues to evolve in future issues of *Connections*.

Thread-Safe Marshaling

Mark Spruiell, Senior Software Engineer

Introduction

Ice simplifies the task of developing distributed object applications, allowing you to focus on building your application and not on the mundane details of marshaling, communication, and interoperability. However, thread safety is one aspect of application development that Ice cannot handle for you completely.

The Ice run time is internally thread-safe, which means that multiple application threads can safely call methods on Ice run-time objects and invoke remote operations without fear of synchronization problems, but only you can ensure that your Ice server is thread-safe. This article describes a thread-safety issue that, if not properly understood, can result in subtle errors in your program.

Dispatching Operations

Before we delve into the problem, let's first review the steps taken by the Ice run time to dispatch an operation, as shown in Figure 1.

Upon receipt of a request message, the Ice run time in the server unmarshals the in-parameters and passes them as arguments to the appropriate method in the servant. The return value and out-param-

eters provided by the servant are marshaled into a reply message that is sent back to the client.

Of particular importance to this article is the marshaling of the operation's results, which is the responsibility of the Ice run time. This is one of those "mundane details" that applications normally needn't be concerned with. In fact, there is no reason to be concerned, provided your object adapter uses a single-threaded configuration.

However, when your configuration allows requests to be dispatched concurrently, there is a subtle implication here that is easily overlooked: the results of the operation may contain references to the servant's private state. Eventually, the Ice run time will marshal the results, but it's possible for a concurrent request to modify the private state before the run time has had a chance to marshal it. And therein lies the problem.

An Example

The following Slice definitions describe a contrived inventory application that illustrates the thread-safety issue:

```
// Slice
module Inventory
{
    struct Location
    {
        string aisle;
        string shelf;
    };

    struct ProductInfo
    {
        string id;
        string desc;
        float cost;
        float markup;
        Location loc;
    };

    interface Warehouse
    {
        ProductInfo getProductInfo(string id);
        void updateCost(string id, float cost,
            float markup);
    };
};
```

Now imagine that we have implemented a `Warehouse` object in a language for which a Slice struct is mapped to a reference type, such as Java. We'll also assume that our server is configured with several threads in its thread pool so that requests can be dispatched concurrently.

The thread-safety issue is depicted in Figure 2, where T1 and T2 represent threads that are dispatching operations on a `Warehouse` servant. T1 invokes `getProductInfo` on the servant and receives a reference to a `ProductInfo` value (a Java object reference) as

Figure 1: Dispatching an operation

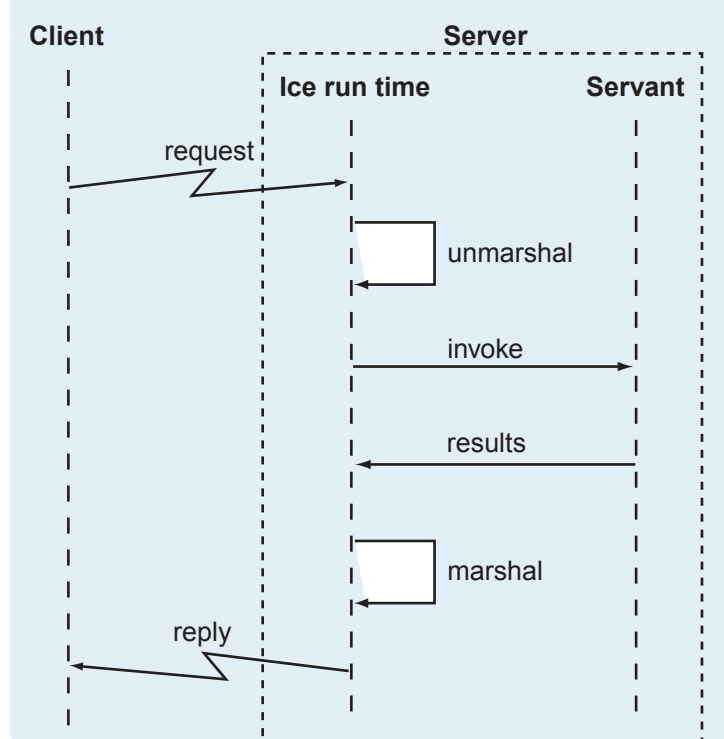
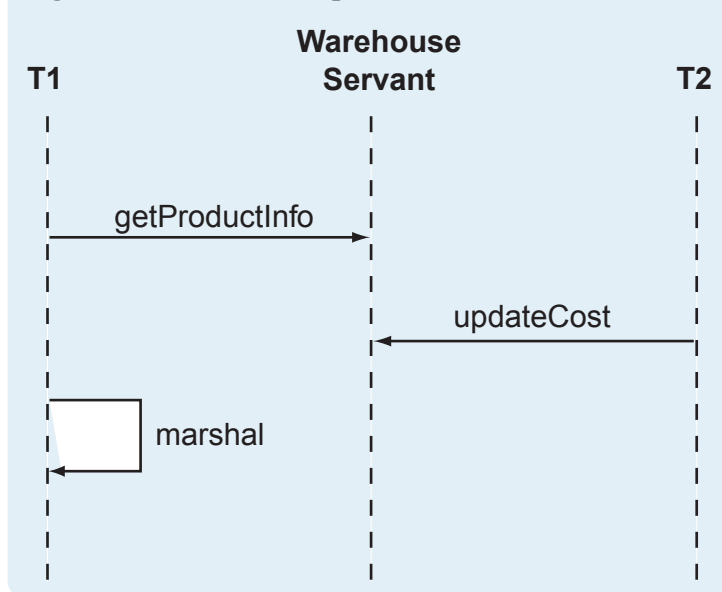


Figure 2: Concurrent requests on a servant



the return value. Next, T2 invokes `updateCost` on the same product. Finally, T1 marshals the `ProductInfo` value.

Depending on the server implementation, it is possible for the `ProductInfo` value held by T1 to be modified by T2 before T1 has a chance to marshal it. Even worse, the `ProductInfo` value may be in an inconsistent state at the time it is marshaled.

For example, here is one implementation of a `Warehouse` servant that exhibits this problem:

```
// Java
public class WarehouseI extends _WarehouseDisp
{
    public synchronized ProductInfo
    getProductInfo(String id, Current curr)
    {
        ProductInfo info =
            (ProductInfo)_products.get(id);
        return info;
    }

    public synchronized void
    updateCost(String id, float cost,
               float markup, Current curr)
    {
        ProductInfo info =
            (ProductInfo)_products.get(id);
        info.cost = cost;
        info.markup = markup;
    }

    private HashMap _products = new HashMap();
}

```

It's not difficult to imagine an unfortunate sequence of events in which T1 begins marshaling the struct and manages to encode the

`cost` member before the thread is interrupted and T2 is scheduled. The `updateCost` operation in T2 is able to modify the `cost` and `markup` members, then T1 is scheduled again and marshals the new value for the `markup` member. The `ProductInfo` struct received by the client contains the old value for `cost` and a new value for `markup`, which could result in some problematic miscalculations.

Synchronizing the methods gives this implementation the appearance of thread safety, but does not actually provide it: `getProductInfo` returns a value by reference to which it no longer controls access. In other words, the synchronization protecting the servant's state lasts only as long as `getProductInfo`; once the method returns, the calling thread must still marshal the return value, but that thread has no way of synchronizing access to it. If it is possible for another thread to modify the value held by the first thread, as in this example, then `getProductInfo` can potentially return a value in a non-deterministic state.

All of the Slice language mappings suffer from this issue, but to varying degrees. In C++, only instances of Slice classes are affected. In Java, C#, Visual Basic, and Python, this problem affects Slice sequences, dictionaries, structs, and classes. (The mapping for structs can be altered in C# and Visual Basic to have value semantics instead of reference semantics; structs using this alternative mapping are not affected.)

Data members of these types are also affected recursively. For example, suppose a Slice operation returns a struct containing a class data member. Although in C++, the struct is not affected, its class data member is still susceptible to the problem.

Solution #1 - Return Copies

One way to solve the problem is to return a copy of the servant's state. Assuming the data is copied using synchronization to protect its integrity, it can be marshaled at some later point by the calling thread without the risk of modification by other threads.

As you might suspect, there are several disadvantages to this strategy. First, excessive copying can have an adverse impact on throughput for performance-sensitive applications. Second, the implementation must return deep copies in order to avoid the same thread-safety issue for nested data members. Finally, writing the code for creating deep copies is both tedious and error-prone, as it requires careful tracking of changes to the Slice definitions.

As an example, here is an implementation of `getProductInfo` that correctly copies the `ProductInfo` struct:

```
// Java
public synchronized ProductInfo
getProductInfo(String id, Current curr)
{
    ProductInfo info =
        (ProductInfo)_products.get(id);
    ProductInfo result = null;
}

```

```

try
{
    result = (ProductInfo)info.clone();
    result.loc = (Location)info.loc.clone();
}
catch(CloneNotSupportedException ex)
{
    assert(false);
}

return result;
}

```

Notice that we are now cloning the `ProductInfo` value, as well as its `Location` member. If additional reference data members are added to the `ProductInfo` struct in the future, we would have to remember to update this method as well as everywhere else we were making a copy of it.

Solution #2 - Copy on Write

Rather than returning a copy of the servant's state, we can make the state immutable. In every operation that modifies the state, we replace the state value with a shallow copy containing the updated value, as shown below:

```

// Java
public synchronized ProductInfo
getProductInfo(String id, Current curr)
{
    ProductInfo info =
        (ProductInfo)_products.get(id);
    return info;
}

public synchronized void
updateCost(String id, float cost, float markup,
           Current curr)
{
    ProductInfo info =
        (ProductInfo)_products.get(id);

    try
    {
        ProductInfo info2 =
            (ProductInfo)info.clone();
        info2.cost = cost;
        info2.markup = markup;
        _products.put(id, info2);
    }
    catch(CloneNotSupportedException ex)
    {
        assert(false);
    }
}

```

In `updateCost`, we clone (shallow copy) the original value, update the members, and then replace the entry in the map. If another thread simultaneously invokes `getProductInfo`, the returned `ProductInfo` object can be safely marshaled by the Ice run time because it is never modified.

This approach is more efficient than the first solution because it doesn't require copying in nonmutating operations, but it still requires discipline in mutating operations to ensure that the immutable quality of the state members is preserved.

Solution #3 - AMD

The thread-safety issue arises because the marshaling of return values and out-parameters happens outside the servant's control. We can regain this control by using Asynchronous Method Dispatch (AMD) instead. First, we must annotate our Slice definition of `getProductInfo` with metadata that directs the translator to use AMD:

```

// Slice
interface Warehouse
{
    ["amd"] ProductInfo getProductInfo(string id);
    void updateCost(string id, float cost,
                   float markup);
};

```

Then we update our servant accordingly:

```

// Java
public synchronized void
getProductInfo_async(
    AMD_Warehouse_getProductInfo cb,
    String id,
    Current curr)
{
    ProductInfo info =
        (ProductInfo)_products.get(id);
    cb.ice_invoke(info);
}

public synchronized void
updateCost(String id, float cost, float markup,
           Current curr)
{
    ProductInfo info =
        (ProductInfo)_products.get(id);
    info.cost = cost;
    info.markup = markup;
}

```

AMD is normally used in situations where the servant needs to delay its response to the client without blocking the calling thread, but we are using it here because it allows us to explicitly marshal the response inside a synchronization block, thereby preventing any other operation from simultaneously modifying the state.

Summary

Thread safety merits careful attention in all but the most trivial of Ice applications. Although the range of affected Slice types varies with language mappings, it is important to keep this issue in mind during development in order to avoid subtle problems after deployment.

Ice and Grid Computing

Bernard Normier, Senior Software Engineer
Benoit Foucher, Senior Software Engineer

According to Webopedia, “grid computing harnesses unused processing cycles of all computers in a network for solving problems too intensive for any stand-alone machine” [1]. Grid computing is now often used in a broader sense: achieving application scalability by exploiting many commodity servers instead of a few big boxes [2].

Ice provides a sound and easy-to-use distributed object infrastructure, and is a smart choice for many kinds of distributed applications, including traditional grid software for High Performance or High Throughput Computing [3]. The MG project [4]—a lightweight Grid for High Energy Physics—is a nice example of how Ice can be adopted to develop practical solutions in this domain.

ZeroC is approaching grid computing from a somewhat different and wider angle: our focus is on the deployment, execution, and monitoring of Ice servers on a large number of computers. These servers may perform long computations requested by a single client, serve thousands of clients, or anything in between. You decide how to exploit the grid, and let Ice handle many of the nitty-gritty details.

The next major Ice release will introduce a new location and deployment service named IceGrid. If you are already familiar with IcePack, don't worry:

IceGrid will provide all of IcePack's functionality and much more. Like IcePack, IceGrid consists of one registry daemon and any number of node daemons. Adding a new computer to your (Ice-based) grid is a simple matter of installing and starting an IceGrid node daemon.

Once you've installed the IceGrid node daemon on a new computer, how do you incorporate it into your application? You first transfer your server binaries to the new computer, and then you make your ap-

plication aware of the new node. IceGrid can dramatically simplify these tasks:

- IceGrid can copy server binaries from a central repository (IcePatch server) to the desired computers. (See Figure 1.)
- Identical servers are easy to deploy on the grid. Using IceGrid's template feature, deploying a new server instance requires minimal effort.
- A set of servers can be configured as replicas of each other. When IceGrid resolves an indirect proxy for an object hosted on replicated servers, it can return a direct proxy with the endpoint of just one server—for example the endpoint of the server with the lowest load—or a direct proxy with the endpoints of several servers. For each set of replicas, IceGrid lets you choose from a number of load-balancing strategies. (See Figure 2.)
- Servers can be pre-deployed on a computer before it joins the grid. When the IceGrid node starts up, the necessary binaries are transferred and the servers are ready to run.

IceGrid nodes provide performance data about their hosts, such as CPU and memory usage, network statistics, and more. This functionality is built into IceGrid, sparing you the need to deploy and configure other software packages, such as SNMP, for this purpose. Applications can use the performance data in a variety of ways: in the implementation of a custom load-balancing strategy, for automatic deployment of new servers when the load reaches a given threshold, to create historical graphs of the grid utilization, and so on.

Figure 1: IceGrid/IcePatch integration

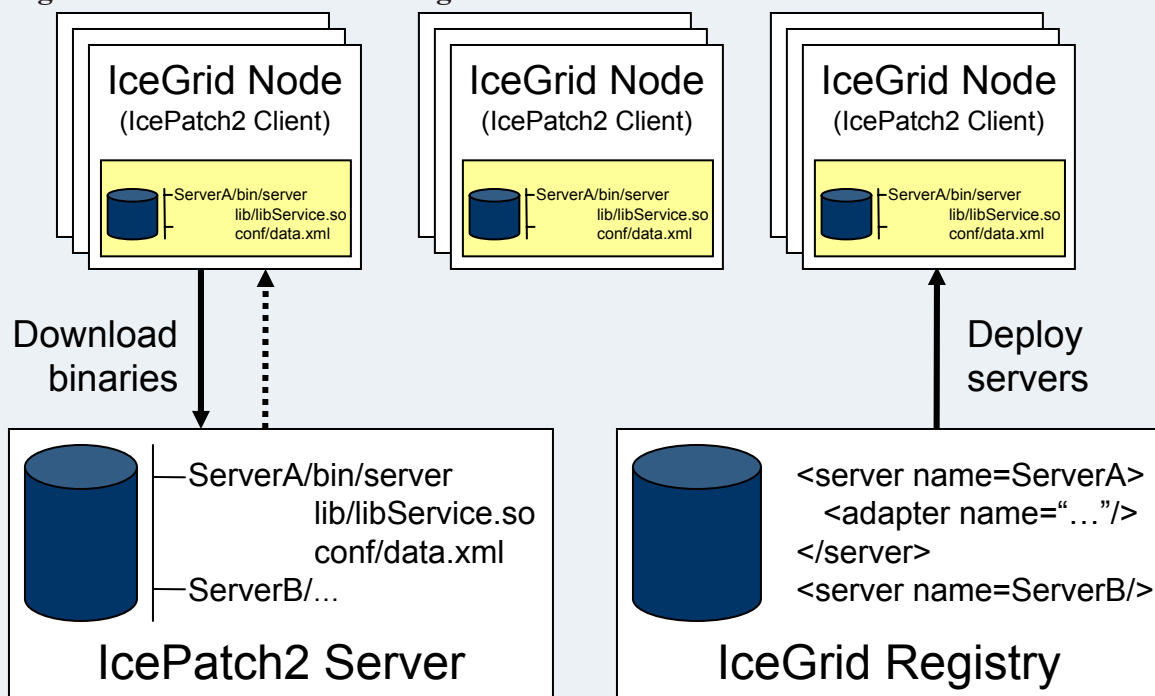
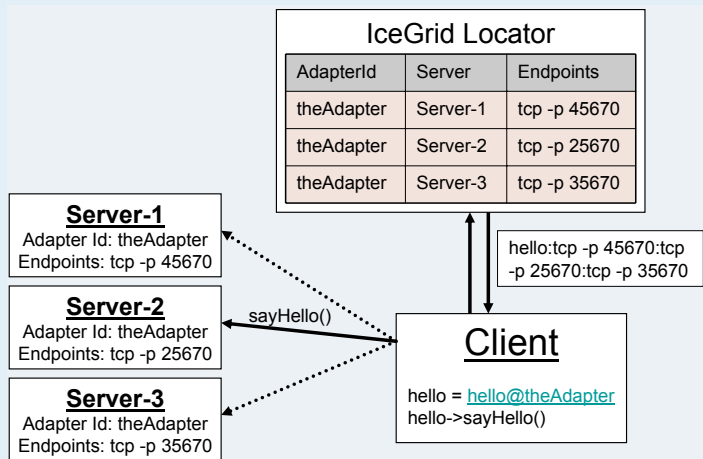


Figure 2: Replication



Finally, IceGrid includes two administrative tools, a command-line utility that is appropriate for simple commands and scripting, and a cross-platform graphical interface suitable for interactive use and server/node monitoring.

Please note that the features outlined above are subject to change before release. We appreciate your feedback: please post your comments on our forum at <http://www.zeroc.com/vbulletin/>.

- [1] Wikipedia, *Grid computing definition*. http://www.wikipedia.com/TERM/g/grid_computing.html.
- [2] eWeek. 2004. *ebay: Sold on Grid*. <http://www.eweek.com/article2/0,1759,1640234,00.asp>.
- [3] University of Wisconsin-Madison. *High Throughput Computing*. <http://www.cs.wisc.edu/condor/htc.html>.
- [4] Nicolai Smirnov. 2005. *MG: Lightweight Grid for HEP*. <http://nsmirnov.home.cern.ch/nsmirnov/mg/mg.html>.

FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about Ice. The questions and answers are taken from our support forum at <http://www.zeroc.com/vbulletin/> and deal with specific problems that developers tend to encounter, and for which the answer may not be readily apparent from reading the documentation. We hope that you will find the hints and explanations in this section useful.

Q: *How can a oneway request be lost?*

Unlike datagrams, which use the unreliable UDP protocol, oneway invocations in Ice use the reliable TCP/IP protocol, just like twoway invocations. (Note that reliable does not mean infallible—see Tip 9 in *Effective TCP/IP Programming* by Jon Snader.)

Because oneway invocations use TCP/IP, you might be tempted to think that they are just as reliable as twoway invocations: for a twoway invocation, it is guaranteed that the request is either delivered or, if it is not delivered (or the client cannot be certain about the delivery status), the client receives an exception.

However, for oneway invocations, it is impossible to provide a guarantee that is as strong as for twoway invocations. The difference is that oneway invocations are unreliable in the face of connection closure (either accidental or deliberate). To understand why, let us first consider how twoway invocations are processed.

When a client sends a twoway request to a server, the Ice run time constructs a protocol message for the request and writes the request data to the client's TCP/IP stack to be sent to the server. The client-side run time then waits until one of the following occurs:

- The run time receives a reply from the server indicating that the request was processed and worked OK.
- The run time receives an exception reply from the server indicating that the request could not be processed. This can happen for a number of reasons. For example, the target object may not exist, or the operation implementation may throw a user exception.
- The run time receives an error from the TCP/IP stack indicating that something went wrong. For example, the server may have deliberately closed its end of the connection, or the network may have been physically damaged (for example, by someone tripping over a network cable and pulling the plug).

It is the third case that is relevant to the difference between twoway and oneway requests.

For twoway requests, the following semantics apply:

- If the server closes a connection deliberately, the client receives a close connection message from the server. The Ice protocol rules ensure that the Ice run time can safely retry any requests for which it has not yet received a response without violating at-most-once semantics. Only if a request cannot be delivered after retrying it does the Ice run time propagate an exception to the application code.
- If the connection is closed forcefully (such as when the server crashes or there is a network problem), the Ice run time either notices the closure because the local TCP/IP stack reports an error, or it times out after a period of inactivity. If the operation is idempotent or nonmutating, the run time retries the failed request. If that attempt also fails, or if the operation is a normal (not idempotent or nonmutating) operation, the run time raises an exception with the application code.

These semantics mean that, for twoway requests, the client application code can rely on getting an exception if something goes wrong; not getting an exception means that the invocation worked.

Now consider the same two connection closure scenarios for oneway invocations.

For oneway invocations, the client-side run time does not wait for a response to a request. Instead, it simply writes the protocol message to the local TCP/IP stack and returns control to the calling application code. This means that, as a rule, the request is still buffered in the client's local TCP/IP stack by the time a oneway invocation completes on the client side. (Note that this also applies for large requests that do not fit into the TCP/IP stack without being split into smaller segments: for such requests, the Ice run time blocks until the TCP/IP stack has accepted all segments, but some of these segments may still be buffered when control returns to the application code.)

In the first scenario, in which the server closes the connection deliberately, there is no way for the client-side Ice run time to notify the application code that a request may have been lost because the thread of control for the request has long since been passed back to the application. This means that the client application code sees a successful completion of a oneway invocation even though the actual request for that invocation never made it into the network. Moreover, subsequent calls may very well succeed, because the Ice run time will transparently open a new connection to the server once the application sends another request (oneway or twoway).

In the second scenario, in which the connection is closed forcefully, there is also no way for the Ice run time to notify the application code, for the same reasons. However, in this case, it is more likely that at least subsequent oneway calls will raise an exception, because it is unlikely that the run time will succeed in establishing

a new connection when it retries (unless you have redundant copies of the server or the server is restarted prior to the retry).

All this means that oneway invocations are unreliable in the face of connection closure and simply cannot provide the same reliability guarantees as twoway invocations. There is little you can do about forceful connection closure, but you can at least take steps with respect to graceful connection closure to improve the reliability of oneway invocations. There are two main considerations: active connection management (ACM) and server shutdown.

Active Connection Management and Oneways

Active connection management is a feature of the Ice run time that allows connections to be reclaimed automatically and transparently: the run time closes connections once they have been idle for some time and re-establishes them as needed. If you must use oneway invocations for efficiency reasons, but cannot afford to silently lose them, you must control ACM explicitly.

On the client side, ACM does not interfere with sending oneway invocations. This is because the run time resets the idle timer for a connection every time a oneway invocation is sent. (The only way ACM could interfere here is if you were to set the timeout so short that a single large oneway invocation cannot be buffered within the ACM timer interval.)

However, on the server side, ACM really gets in the way: the ACM thread in the server can close a connection at any time without warning, leading to the loss of oneways we mentioned previously. It follows that, if you need reliable oneway invocations, you must disable ACM on the server side. You can do this by setting the property `Ice.ACM.Server` to zero.

Server shutdown and Oneways

If the server shuts down deliberately (by calling `shutdown` on its communicator), the Ice run time closes all incoming connections for that communicator (once all executing operation invocations have finished). Of course, this means that clients can lose buffered oneway invocations just as if ACM had closed the server end of a connection. If your application cannot tolerate the loss of oneway invocations, you must somehow make sure that the server does not shut down while clients can have buffered oneway requests. How to do this is up to your application—a common solution is to have the server notify its clients that it is about to go away. Each client then sends a twoway confirmation message back to the server. Once the server has received (and replied to) all of these confirmation messages, it can shut itself down. And, conversely, completion of the confirmation message in the client without an exception guarantees that all previously buffered oneway invocations have been sent because a twoway invocation cannot “overtake” oneway invocations that were buffered previously.

Note that all of this is necessary only in the following cases:

- The server is restarted once shut down, and manages to both shut down and restart in between two successive oneway invocations by a client.
- You have redundant copies of the server and proxies for objects in these servers have multiple endpoints.

If neither of these cases applies, the client will at least receive an exception when it tries to send subsequent oneways, and therefore receive an indication that something is not working correctly.

Batch Oneways

Everything we said about the reliability of oneways also applies to batch oneways. The only difference is that, if a batch oneway message is lost, all invocations in the same batch are lost with it.



Why can oneway requests block?

If an operation has a void return type and does not have out-parameters or an exception specification, you may be tempted to simply call the operation via a oneway proxy, thinking that this makes it impossible for the client to block when it invokes the operation. However, this is not the case: oneway operations can indeed block.

This is actually a Good Thing™. If oneway invocations could not block, the caller would never know about certain error conditions, including error conditions that mean that invocations can never work at all (such as when the proxy contains an incorrect address for the server).

There are two scenarios that can cause a oneway invocation to block the caller:

Connection Establishment

If no connection to the target server for an invocation is currently established, the Ice run time will transparently open a connection (whether the call is oneway or twoway). Connection establishment forces the Ice run time to wait for a connection validation message from the server, as required by the Ice protocol. If the server is slow to respond, has run out of threads to process its incoming connection request, or misbehaves in other ways, an invocation (including a oneway invocation) can block. If you set a timeout, the call will throw a `ConnectionTimeoutException` once the timer expires; without a timeout, the call can block indefinitely. Regardless, connection establishment is identical for oneway and twoway operations and can block the client.

Note that you can mitigate the problem somewhat by forcing a connection to be established initially, for example, by disabling ACM and sending an `ice_ping` before you send a oneway invocation. However, because connections may be closed due to networking problems, this only makes it less likely for a oneway call to block, but does not prevent it, because, during a retry of a failed invocation, the Ice run time may attempt to re-establish a connection and block at that point.

TCP/IP Buffer Limitations

The client's local TCP/IP stack has a limited amount of buffer space to accept data. If a oneway request is too large to fit into the remaining TCP/IP buffer space, the kernel suspends the caller in its `write` system call on the socket until enough buffer space becomes available. The remaining buffer space can be consumed by previously buffered requests or even by a single request, if it is large enough. Either way, a oneway invocation can block until the TCP/IP stack has removed enough data from its buffers for the currently executing request to be buffered, at which point the oneway invocation returns the thread of control to the application code.

How to Avoid Blocking

As of Ice 3.3, asynchronous (AMI) invocations can no longer block the caller. This means that if you invoke an operation (whether oneway or twoway) asynchronously, the call will never block, even if a connection cannot be established immediately, a DNS look-up is slow, or the TCP/IP transport buffers are full.

If you asynchronously invoke a oneway operation, the `ice_response` method of your async callback object simply will never be called. (However, errors encountered during asynchronous oneway invocations, such as failure to establish a connection, are reported via `ice_exception` as usual.)



How can oneway requests arrive out of order?

Sequentially-sent oneway requests cannot arrive out of order, but the server may dispatch them out of order.

Assuming that oneway requests are not sent in parallel by multiple threads, and that all oneway requests use the same connection to the server (which is usually the case, unless different protocols or timeouts are used), then the client-side Ice run time ensures that the protocol messages for these requests are sent in order.

However, if the server employs a thread pool, then the server-side Ice run time will dispatch oneway requests in parallel, up to the maximum number of threads that is configured for the thread

pool. It is therefore possible for oneway requests to be dispatched in an order that differs from the order in which they were received, depending on how the operating system schedules threads.

Out-of-order dispatch cannot happen with twoway invocations. For sequentially-sent twoway invocations, the client always waits until it has received a response from the server for an outstanding twoway before it sends another twoway. Since the server only sends a response once it has completed dispatching of the twoway call (or at least started dispatching, if the server uses asynchronous message dispatching), twoways cannot be dispatched out of order.

There are three ways to avoid out-of-order dispatching of oneway requests. The first way is to simply configure only one thread for the server-side thread pool. With only one thread, no requests can be dispatched in parallel. The second option is to force the thread pool to dispatch requests serially by setting one of the properties `Ice.ThreadPool.Client.Serialize`, `Ice.ThreadPool.Server.Serialize`, or `adapter.ThreadPool.Serialize`. Finally, you can use batched oneways. All oneway requests from a single batch are guaranteed to be dispatched in order by the same server thread, regardless of the number of threads in the thread pool.