What is RPC?

My last editorial caused quite a lively discussion about SOAP’s performance. I recommend that you read Jim Webber’s response as well as my reply and make up your own mind about this issue. In any case, I’d like to thank the participants for their feedback.

The discussion about SOAP performance brought up an unrelated but very interesting topic, namely the nature of RPC (remote procedure call) and how SOAP compares to CORBA or Ice with respect to the use of RPC.

Michi believes that every request/response system, including SOAP, is RPC. Wikipedia seems to agree with him. Others disagree, believing instead that something is RPC only if, at the API level, the request/response has the “look and feel” of a local procedure call. Since SOAP can be used with other request/response styles, such as using separate message queues for requests and responses, they claim that SOAP is not RPC.

I’m not interested in debating the definition of RPC. However, if SOAP is not RPC, then neither are Ice and CORBA. Both Ice and CORBA offer many different invocation and dispatch styles, including asynchronous method invocation (where separate threads or callbacks handle responses), asynchronous method dispatch (to decouple sending the response from receiving the request on the server side), time-independent invocations, invocations using message and event services, and so on. The point is, no matter how you define RPC, SOAP offers nothing that Ice or CORBA do not offer as well.

Quite independent from invocation style though, the fact is that all distributed systems must deal with issues such as latency, concurrency, network reliability, and error recovery. No middleware can make these problems go away, regardless of whether you use SOAP, Ice, CORBA, or plain sockets.

However, what kind of solutions you can apply to solve these problems heavily depends on the middleware you use. What it eventually comes down to is what tools your middleware provides for interface design, how it allows you to distribute interface implementations over physical servers and at what granularity, whether it supports both static and dynamic typing, and how much of the type checking it does for you as opposed to making your application code responsible for it.

There are many political arguments for SOAP, and it’s hard to win these. But when it comes to technical arguments, SOAP is mostly smoke and mirrors. At ZeroC, we are admittedly very technical and we leave the big view about how technology markets work to others. Instead, we concentrate on providing solutions that solve real problems for real customers. It’s as simple as that.

Marc Laukien
President
ZeroC, Inc.

Issue Features

Introduction to IcePack

In this article, Matthew Newhook and Benoit Foucher show you how you can use IcePack to automate the chat application so that servers are started on demand and in the correct order, as well as how to automate the deployment of the application for different environments.

Ice-E: A Preview

Later this year, ZeroC will release Ice-E, a version of Ice for embedded C++ and Java applications. This article provides a brief overview of Ice-E.

Avoiding Deadlocks, Part II

In this second part of his two-part article, Bernard Normier shows you how you can save yourself having to debug deadlock situations by making sure they cannot arise in the first place.

Contents

Introduction to IcePack .................................................. 2
Ice-E: A Preview .......................................................... 10
Avoiding Deadlocks, Part II .......................................... 11
FAQ Corner ................................................................. 19
Introduction to IcePack

Matthew Newhook and Benoit Foucher, Senior Software Engineers

Introduction

In the article “An Introduction to IceStorm” in issue 3 of Connections, we added support for multiple chat rooms through the use of IceStorm. Unfortunately, starting and shutting down all of the servers necessary to run the chat application has now become a chore. This article explores the use of IcePack to make the management and deployment of the chat application much easier. (This article introduces IcePack without meaning to be exhaustive. Please see the Ice manual for full details.)

Why IcePack?

IcePack makes the process of developing, configuring, deploying, and managing applications significantly easier than doing the same thing manually. It provides

- a locator service to locate objects and ensure location transparency
- a server activation and monitoring service to manage server processes
- a sophisticated server deployment mechanism

IcePack itself consists of three major components as shown in Figure 1: IcePack Components. (An IcePack domain is a logical concept used for organizing your application deployment.)

**Figure 1: IcePack Components**

- IcePack registry daemon
  The registry daemon manages information about the applications that are deployed in a particular domain. Only one instance of the registry should be deployed per domain.
- IcePack node daemon
  The node daemon is responsible for on-demand activation and monitoring of Ice servers. You can deploy as many nodes as needed (typically, one per host).

**Figure 2: IcePack Endpoint Configuration**

- IcePack administration tool
  The administration tool is a command-line utility for configuring applications deployed in a domain.

For this article, since we are only deploying our chat application on one node, we will run the registry and the node daemons collocated in one process. To do this, we set the configuration property:

```
// IcePack Configuration
IcePack.Node.CollocateRegistry=1
```

The IcePack registry defines several endpoints. As shown in Figure 2: IcePack Endpoint Configuration, the client endpoints must be accessible to both clients and servers, whereas the server endpoints need be accessible only to the servers that are deployed in the IcePack domain. The admin endpoints must be accessible to the IcePack admin tool or any programs accessing the IcePack::Admin interface. The internal endpoints must be accessible to deployed IcePack nodes.

Note that if you are running the registry and the nodes in an insecure environment, you should use SSL for the server, internal, and admin endpoints. Otherwise, a malicious application could connect to these endpoints and re-configure your deployment, hijack your servers, or start and stop servers of your deployed application.

The client endpoint is a well-known endpoint and must have a fixed port. This endpoint is used by the Ice run time in a client to contact the Ice locator facility. The run time reads the value of the Ice.Default.Locator property to obtain a proxy to this endpoint. On the other hand, the server, admin, and internal endpoints don’t need to be configured with a fixed port, which raises an interesting question: if the admin endpoint does not need a fixed port, how is the IcePack admin tool able to communicate
with the registry? The answer is quite simple: it uses the locator facility. The admin endpoints are registered with the locator, and the IcePack admin tool resolves the `IcePack::Admin` object with the help of the locator just as for any other Ice client.

Our IcePack endpoint configuration is as follows:

```plaintext
// IcePack Configuration
IcePack.Registry.Client.Endpoints=default -p 12000
IcePack.Registry.Server.Endpoints=default
IcePack.Registry.Internal.Endpoints=default
IcePack.Registry.Admin.Endpoints=default
```

All clients that use IcePack (such as `icepackadmin`) must have the `Ice.Default.Locator` configuration property correctly configured. (Since we are running everything on a single host, the host is not configured in the proxy.)

```plaintext
// IcePack Client Configuration
Ice.Default.Locator=IcePack/Locator:default -p 12000
```

### Server Management

First, we recap the process that must be followed to manually start up the chat server.

1. Start IceStorm.
2. Start the Chat Session server.

Upon start-up, the chat server connects to the IceStorm server. The Glacier2 router connects to the chat server to communicate with the session manager and the password validation objects. Therefore, the chat server has a start-up dependency on the IceStorm server, and the Glacier2 router has a dependency on the chat server. For a more complex application, adding and administering additional services and servers with their own set of dependencies quickly becomes difficult, particularly if the servers are distributed over multiple hosts.

---

**INTRODUCTION TO ICEPACK**

IcePack automatically manages your servers’ life cycle. When an Ice object that resides in a particular server is first accessed by a client, IcePack can start the server automatically; this is known as on-demand activation. On-demand activation makes server management significantly easier because it eliminates the need to manually start and stop servers. See *Figure 3: Starting a Server* for an interaction diagram of this process.

### Binding

Up to now, the chat application has used direct binding. Direct binding means that the endpoint details (host, protocol, and port) are fully specified in the proxy. The server must be sure to create any object adapters using the same endpoint details as those specified in any relevant proxies. For example, without IcePack, the chat server configuration file (`config.server`) contains

```plaintext
// Chat Server Configuration
ChatServer.Endpoints=tcp -h 127.0.0.1 -p 10001
IceStorm.TopicManager.Proxy=IceStorm/TopicManager:tcp -p 10000
```

`ChatServer.Endpoints` contains the endpoint details for the `ChatServer` object adapter. This endpoint runs on a fixed port. Without indirect binding, all proxies to objects hosted by this object adapter must specify the exact same endpoint information. `IceStorm.TopicManager.Proxy` also contains endpoint details. In this case, it specifies that the server can be located on localhost, on port 10000. The IceStorm server configuration has to contain the exact same endpoint configuration as the proxy. For simple deployments, it is not difficult to maintain this information. However, once an application becomes larger or is deployed across multiple hosts, it becomes much more problematic, especially if you need to make changes to the host or ports assignments.

IcePack enables applications to use indirect binding. Indirect binding adds a layer of indirection between the client and the server. The client no longer needs to specify the endpoint details for a proxy. Rather, the endpoint details are determined dynamically at runtime by the Ice run time. See *Figure 4: Indirect Binding* for an interaction diagram of this process.

Another big advantage of using indirect binding is that, in contrast to CORBA, the proxy contains no endpoint information at all. It is, for example, possible to store a proxy in a database, and then at a later point continue to use the proxy—even if the locator has moved location! As long as the application is configured with the correct location of the locator, the proxies all continue to work.

---

**Figure 3: Starting a Server via IcePack**

![Diagram of starting a server via IcePack](image-url)
Using IcePack, applications can locate objects in three ways.

1. Object identity
   Given the object identity, the application queries IcePack and locates the given object.

2. Object type
   Given an object type-id, IcePack returns all objects that implement this type.

3. Object identity @ adapter id
   Given the object identity and an adapter id, the application queries IcePack and locates the given object adapter.

**Server Deployment**

Deployment descriptors make your application very easy to manage: you describe a deployment once and deploy many times with potentially different deployment scenarios, such as deployment for a test suite and deployment for production. A deployment descriptor is an XML file that describes the structure of your application. Deployment of your application does the following:

- creates server configuration files
- initializes any required Freeze database environments
- registers servers with the appropriate IcePack node daemon
- adds object adapters to the IcePack registry
- adds well-known objects and their identities to the IcePack registry

**Preparing the Chat Session Server**

Strictly speaking, the server can be managed by IcePack as it is. However, in order to fully take advantage of IcePack, we must make a few changes. IcePack generates configuration files from a server’s deployment descriptor, so we must remove the hard-coded configuration file.

```cpp
// C++
int main(int argc, char* argv[])
{
    ChatSessionServer app;
    return app.main(argc, argv);
}
```

Since we want to use indirect binding, we can remove the IceStorm TopicManager proxy from the configuration file and instead look up the object by identity. The code originally was:

```cpp
// C++
string str = communicator->getProperties()->getProperty("IceStorm.TopicManager.Proxy");
_manager = IceStorm::TopicManagerPrx::checkedCast(communicator->stringToProxy(str));
```

The code now becomes:

```cpp
// C++
_manager = IceStorm::TopicManagerPrx::
    checkedCast(communicator->
        stringToProxy("IceStorm/TopicManager"));
```

Under the covers this locates the object by identity using IcePack. It is equivalent to the following code:

```cpp
// C++
IcePack::QueryPtr query = IcePack::Query::
    uncheckedCast(communicator->
        stringToProxy("IcePack/Query"));
Identity id;
id.name = "IceStorm/TopicManager";
_manager = IceStorm::TopicManagerPrx::
    checkedCast(query->findObjectById(id));
```

How does the application know how to locate the IcePack registry? This is configured in the configuration property Ice.Default.Locator and is provided automatically by IcePack to any IcePack managed servers.

These are all the changes that are necessary to enable the chat server to be an IcePack-managed server. Next we need to write the deployment descriptor.

**Deployment Descriptors**

An IcePack deployment descriptor contains a description of:

- nodes on which the servers are deployed
- servers that run on the nodes
- IceBox services that are hosted by a server, if applicable
- object adapters, and well known objects and their type hosted by the servers and services
Since we want to run our initial deployment on only one host, our initial version of the deployment descriptor is as follows:

```xml
<icepack>
  <application name="ChatApplication">
    <node name="node"> ... </node>
  </application>
</icepack>
```

All deployment descriptors are contained within an `icepack` element. The `application` element is used to contain the entire application deployment. It has one attribute, the name of the application. The `application` element contains one `node` element per physical host in the deployment. The `node` element contains one attribute, the name of the node. You might be tempted to choose node names that correspond to host names of computers. Although using the host name is convenient, it’s not a good idea because, during the lifespan of your application, you might want to migrate a node to another machine with a different host name. It is therefore better to use abstract node names that describe the role of a node, rather than physical host names.

Next we must decide what servers we need to deploy for the full chat server application. What servers are currently run manually?

- Glacier2
- IceStorm
- Chat Server

Obviously, IceStorm and the chat servers should be managed by IcePack, but what about Glacier2? In the current configuration, IcePack could manage Glacier2. However, for many applications (including this one at a later point), the client will want to authenticate the server identity through an SSL certificate. In the absence of some scheme for providing the SSL certificate pass phrase, Glacier2 must be started manually so the operator can enter the pass phrase on the command line.

Now we can write the deployment descriptions of the servers themselves. Here is the XML fragment which represents the chat session server:

```xml
<server name="ChatSession" kind="cpp"
  exe="./server" activation="on-demand">
  <adapters>
    <adapter name="ChatServer" endpoints="tcp">
      <object identity="ChatSessionManager" type="::Glacier2::SessionManager"/>
      <object identity="verifier" type="::Glacier2::PermissionsVerifier"/>
    </adapter>
  </adapters>
  <properties>
    <property name="Ice.ThreadPool.Client.Size" value="4"/>
  </properties>
</server>
```

The `server` element describes a server to be deployed on an IcePack node. This element contains a number of attributes: the name of the server, the kind (or type) of the server (`cpp` in this case since this is a C++ server), the path name of the executable, and the server activation mode.

The `adapters` element identifies the object adapters contained in the server. Each object adapter is described by an `adapter` element. The `adapter` element contains two attributes: the name of the object adapter and the endpoints for the adapter.

The `adapter` element contains a list of well-known Ice objects hosted by this adapter. The `object` element also contains two attributes: the stringified object identity and the fully-scoped Slice type of the object.

The `properties` element contains one or more property elements. Each `property` element generates a configuration property for the server. The `property` element contains attributes that define the property’s name and its value.

Next we examine the deployment descriptor of the IceStorm service.

```xml
<server name="IceStorm" kind="cpp-icebox"
  endpoints="tcp -h 127.0.0.1"
  activation="on-demand">
  <service name="IceStorm"
    entry="IceStormService,21:create">
    <dbenv name="${service}"/>
    <adapters>
      <adapter name="${service}.TopicManager" endpoints="tcp">
        <object identity="${service}/TopicManager" type="::IceStorm::TopicManager"/>
      </adapter>
      <adapter name="${service}.Publish" endpoints="tcp"/>
    </adapters>
  </service>
</server>
```

In this case, the `server` element kind attribute contains `cpp-icebox` since the IceStorm server is an IceBox server, and the `endpoint` attribute configures the endpoint for the IceBox administrative interface.

The `service` element describes each service that is deployed into the IceBox server. It contains two attributes: the name of the service and the service entry point.

The `dbenv` element causes the deployer to create and configure a Freeze database environment. It contains one attribute that defines the name of the database environment. In this case, the name refers to a defined variable by using the `{$}` syntax. The `$service` variable is defined by the `service` element to contain the name of the service.
The full deployment descriptor for the entire chat application is as follows:

```xml
// XML
<icepack>
  <application name="ChatApplication">
    <node name="node">
      <server name="IceStorm" kind="cpp-icebox">
        <endpoint tcp -h 127.0.0.1 activation="on-demand">
          <service name="IceStorm">
            <entry>IceStormService,21:create</entry>
          </service>
        </endpoint>
      </server>
      <server name="ChatSession" kind="cpp">
        <property name="Ice.ThreadPool.Client.Size" value="4"/>
        <adapter name="ChatServer">
          <object identity="ChatSessionManager" type="::Chat::ChatSession"/>
          <object identity="verifier" type="::Glacier2::PermissionsVerifier"/>
        </adapter>
      </server>
    </node>
  </application>
</icepack>
```

Before testing this deployment, a couple of changes need to be made to the Glacier2 configuration file. Since IcePack now manages the session manager and permissions verifier Ice objects, their proxy properties must be changed:

```xml
// Glacier2 Configuration
Glacier2.SessionManager=ChatSessionManager
Glacier2.PermissionsVerifier=verifier
```

Furthermore, Ice.Default.Locator property must be defined so that IcePack can locate these objects.

```xml
// Glacier2 Configuration
Ice.Default.Locator=IcePack/Locator:default -p 12000
```

OK, we're ready to try it out. To test this deployment:

- Start IcePack.
  ```bash
  > icepacknode --Config=config.icepack
  ```
- Populate the deployment information.
  ```bash
  > icepackadmin --Config=config.icepack -e "application add 'server_app.xml'"
  ```
- Start Glacier2.
  ```bash
  > glacier2router --Config=config.glacier2
  ```

Now you can connect a client to the server. The chat server and the IceStorm server will be started on demand. The deployment is as shown in Figure 5: Initial Deployment.

### Figure 5: Initial Deployment

![Initial Deployment Diagram]

**IceBox**

We can reduce the number of processes in the deployment through the use of IceBox. IceBox is an implementation of the Service Configurator pattern for Ice services. IceBox itself is an Ice server that allows a single physical process to host multiple Ice services. Without IceBox, you are forced to make deployment decisions at implementation time. Conversely, with IceBox, each logical server is implemented and packaged into a service. At deployment time you can decide where and how to deploy the services. Some key advantages are:

- Services loaded into one process can take use collocation optimization, which is significantly faster than over-the-wire invocations.
- Resources (such as thread-pools) can be shared.
- Multiple Java services can share the same JVM.

You should read the Ice manual for more information on the advantages of using IceBox.

Creating an IceBox service is a simple process. The first step is to write an implementation of the `IceBox::Service` interface. The interface is as follows:
The set of services that are loaded into an IceBox server are configured through the IceBox server's configuration file, which is generated by IcePack in the deployment process. On server start-up, IceBox calls `start` on each configured service, and `stop` on shutdown. The implementation of `start` must set up any object adapters and Ice objects that are provided by the service; `stop` must deactivate any object adapters and do any necessary clean-up on all Ice objects.

Our implementation will reuse the code that already exists in the `ChatSessionServer` class. The main difference is that we do not call `Communicator::waitForShutdown`, which is taken care of by IceBox instead. The implementation is as follows:

```cpp
// C++

class ChatSessionServiceI : public ::IceBox::Service
{
public:
    virtual void start(
        const string& name,
        const CommunicatorPtr& communicator,
        const StringSeq& args)
    {
        _adapter = communicator->createObjectAdapter("ChatServer");
        _adapter->add(
            new DummyPermissionsVerifierI,
            stringToIdentity("verifier");
        _adapter->add(
            new ChatSessionManagerI(communicator),
            stringToIdentity("ChatSessionManager"));
        _adapter->activate();
    }

    virtual void stop()
    {
        _adapter->deactivate();
    }

private:
    ObjectAdapterPtr _adapter;
};
```

Since this is a C++ IceBox service, you must implement an entry point that is called by IceBox when creating an instance of your service upon server startup. The implementation follows:

```cpp
// C++

extern "C"
{

    CHATSERVICE_SERVICE_API ::IceBox::Service*
    create(CommunicatorPtr communicator)
    {
        return new ChatSessionServiceI;
    }
}
```

Now we'll change the deployment descriptor. This time we'll deploy the entire application into one IceBox server.

```xml
<icepack>

<application name="ChatApplication">
    <node name="node">
        <server name="ChatServer" kind="cpp-icebox"
            endpoints="tcp -h 127.0.0.1"
            activation="on-demand">
            <service name="IceStorm"
                entry="IceStormService,21:create">
                <properties>
                    <property name="Ice.ThreadPool.Client.Size" value="4"/>
                </properties>
                <adapters>
                    <adapter name="${service}.TopicManager" endpoints="tcp">
                        <object identity="${service}/TopicManager" type="::IceStorm::TopicManager"/>
                    </adapter>
                    <adapter name="${service}.Publish" endpoints="tcp"/>
                </adapters>
            </service>
            <service name="ChatSession"
                entry="ChatSessionService:create">
                <properties>
                    <property name="Ice.ThreadPool.Client.Size" value="4"/>
                </properties>
                <adapters>
                    <adapter name="ChatServer" endpoints="tcp">
                        <object identity="ChatSessionManager" type="::Chat::ChatSession"/>
                        <object identity="verifier" type="::Glacier2::PermissionsVerifier"/>
                    </adapter>
                </adapters>
            </service>
        </server>
    </node>
</application>
```

The process to test this is the same as last time. In order to make testing simpler you should shut down the `icepacknode` and clean the database records.

```bash
> rm -rf db/node/* db/registry/*
```
The deployment is now as shown in Figure 6: IceBox Deployment.

**Figure 6: IceBox Deployment**

![IceBox Deployment Diagram]

**Using Includes**

The descriptor that we have written is quite straightforward. However, it lacks flexibility. What will happen, for example, if you want to deploy two instances of the chat service? With the current setup, you’d have to copy and paste text, and then change the variable values appropriately. Imagine that, later, you want to enable some debug configuration settings. Since you now have two copies of the deployment descriptor (one for each instance of the chat service), you now need to edit two sets of descriptors, instead of just one. Obviously, the more servers you deploy using this method, the bigger the problem becomes. Clearly there has to be a better way.

The solution is to use included files. To do this, we place the deployment descriptor for the service in a separate file. In this example, we use `session.xml`.

```xml
// XML
<icepack>
  <include descriptor="session.xml"/>
</icepack>
```

We can create a similar definition for IceStorm, in `icestorm.xml`. We can use these descriptors in the server deployment descriptor as follows.

```xml
// XML
<icepack>
  <include descriptor="icestorm.xml"/>
  <include descriptor="session.xml"/>
</icepack>
```

To deploy two session services, we simply use two includes of `session.xml`. However, with the definition above, this isn’t actually possible because `include` element simply substitutes the included text. The result of the two includes would be to deploy two exact copies of the chat service, and we would end up creating two services with the same name, `ChatSession`, which is a wrong. (Even if this would work, it would also create distinct objects with the same identity (`ChatSessionManager` and `verifier`), which is also wrong). The solution to use variables that are later substituted into the service definition:

```xml
// XML
<include name="ChatSession1" descriptor="session.xml"/>
<include name="ChatSession2" descriptor="session.xml"/>
```

This defines a new variable `name`, which can be used in the descriptor using the syntax `${name}`, so we can change the descriptor as follows:

```xml
// XML
<icepack>
  <service name="${name}" entry="ChatSessionService:create">
    <properties>
      <property name="Ice.ThreadPool.Client.Size" value="4"/>
    </properties>
    <adapters>
      <adapter name="ChatServer" endpoints="tcp">
        <object identity="${name}-ChatSessionManager" type="::Chat::ChatSession"/>
        <object identity="verifier" type="::Glacier2::PermissionsVerifier"/>
      </adapter>
    </adapters>
  </service>
</icepack>
```
For projects of any complexity, it is generally a good idea to create individual descriptors for each node, server, and service. Although this makes the descriptors somewhat more difficult to understand, the flexibility and ease of maintenance make this setup much easier to manage. (The next article in this series will utilize such a setup for its deployment descriptors.)

### Using Deployment Targets

Imagine that you want to add some debug properties to your chat server deployment descriptor. For instance, you might want to trace network connection activity. To do this you might do the following:

```xml
// XML
<icepack>
  <service name="${name}" entry="ChatSessionService:create">
    <properties>
      <property name="Ice.ThreadPool.Client.Size" value="4"/>
    </properties>
    <adapters>
      <adapter name="ChatServer" endpoints="tcp">
        <object identity="${name}-ChatSessionManager" type="::ChatContract::ChatSession"/>
        <object identity="${name}-verifier" type="::Glacier2::PermissionsVerifier"/>
      </adapter>
    </adapters>
    <properties>
      <property name="Ice.Trace.Network" value="1"/>
    </properties>
  </service>
</icepack>
```

This works nicely. However, what if you want to turn off the tracing? You would need to edit the deployment descriptor and then re-deploy. We can easily come up with situations in which this approach becomes more difficult. For example, consider a need to use a different password database for testing and production deployments. The connection string for the database is contained in the application configuration, which comes from the deployment descriptor. It follows that, to support the two deployment scenarios, we need to have separate deployment descriptors, which again causes a maintenance problem. Fortunately, IcePack provides an elegant solution for this situation: deployment targets. Consider again the debug/non-debug scenario. You can create a deployment target called "debug", and then turn on debugging information when this target is deployed.

```xml
// XML
<target name="debug">
  <properties>
    <property name="Ice.Trace.Network" value="1"/>
  </properties>
</target>
```

Now when you configure IcePack you can tell it to use this target.

```bash
> icepackadmin --Config=config -e "application add server_app.xml debug"
```

Furthermore, you can take a deployed application and change its targets.

```bash
> icepackadmin --Config=config -e "application update server_app.xml debug"
```

This means that you can maintain one set of deployment descriptors, but still have different deployment and server configurations.

### Summary

This article showed you how to get started with IcePack. Even though the initial learning curve looks a bit steep, we encourage you to give IcePack a try. IcePack allows you get rid of other things you would otherwise need to build, such as start-up and shut-down scripts and you will find that, especially for more complex applications, using IcePack provides you with a great deal of flexibility.
Ice-E: A Preview

Marc Laukien, President & Founder

Overview

“Embedded Ice” (“Ice-E”) is a new product currently in development at ZeroC. As its name implies, Ice-E is specifically designed for embedded systems and therefore features a very small memory footprint. With IceGrid (presented in Connections issue number 2), we extend Ice into the world of the very large: high-performance computing using large computer grid systems. With Ice-E, we round out the Ice product line towards the lower end, the world of the very small: cell phones, PDAs, embedded controllers, and many more.

The estimated release date for the product is September 2005, and the initial release will support the C++ and Java language mappings. Ice-E for C# is planned for a future release.

Ice-E supports a subset of Ice. As such, some features have been removed, and some have been made optional. However, as there is no one-size-fits-all for embedded systems, ZeroC will offer customized versions to commercial customers that add or remove features from the default Ice-E distributions.

Ice-E is fully interoperable with Ice, with the only exception being Objects-by-Value. We considered the supporting code for Objects-by-Value too large to be suitable for embedded systems. Furthermore, the first release will support only TCP/IP (for example, over regular Ethernet cards, Bluetooth, or wireless standards such as GPRS or EDGE). Upon request, ZeroC can add other protocols for commercial customers. Future versions of Ice-E will also incorporate real-time capabilities.

Ice-E for C++

Ice-E for C++ initially supports the following platforms:

- Windows CE 4.2 (Pocket PC/Smartphone)
- Windows XP
- Linux

Of course, ZeroC can also provide versions for other platforms for commercial customers.

Ice-E for C++ provides two separate libraries. The first is for pure clients, which allows the creation of statically linked Ice clients as small as 150KB (Windows XP, VC++ 7.1). The second library is for mixed client/servers; a minimal statically linked server is approximately 190KB in size.

Ice-E for C++ provides the full C++ language mapping, including exceptions and STL. Source code is compatible with the full Ice for C++, as long as none of the removed features are used.

Ice-E for Java

Ice-E for Java supports J2ME CLDC 1.1 + MIDP 2.0, as well as the regular J2SE. (J2ME is a subset of J2SE 1.3.) MIDP 2.0 is widely supported in embedded devices, such as Windows CE Smartphones, Nokia cell phones, and many more.

In the current prototype, a minimal Ice-E for Java client can be as small as 80KB, a minimal server about 90KB. We expect to be able to lower the memory footprint even further in the final release.

An Example

Our chat demo application serves as a perfect example to demonstrate how Ice-E can be used on embedded devices. Figure 1 shows a chat client written with Ice-E for C++, running on an HP iPAQ PXA270. This and other demos will be included in the Ice-E distributions.

Figure 1: Chat Client on iPAQ
Avoiding Deadlocks, Part II

Bernard Normier, Senior Software Engineer

Introduction

Last month, the first part of this article presented various kinds of distributed deadlocks and their solutions. You may remember that some deadlocks are quite insidious: a deadlock that strikes rarely can be very difficult to find. This month, we will take a step back and look for a strategy that prevents deadlocks in the first place—prevention is better than a cure.

Deadlock-avoidance is not a sexy feature—you will never see a brochure touting how deadlock-free a product is—and is often put on the back burner. Real features are the priority; deadlocks can be fixed when they occur. Unfortunately, this “wait ‘til it breaks” attitude can be very dangerous for deadlocks; a poorly designed application can embed an infinite number of such bugs. You don’t want to discover during system testing or deployment that your application needs a major overhaul because it “hangs” now and then.

A consequence of the low status of deadlock-avoidance is the need for a simple strategy. Nobody wants to write lots of code or expend lots of brainpower on a feature that will never be mentioned. For example, putting every single remote call on a big diagram to eliminate loops is not a practical solution because it’s hard to maintain and it’s intrusive—as a programmer, I don’t want to ask permission each time I want to add another remote call. Simplicity also helps getting the word out, which is critical here: the chosen strategy must be widely followed to be successful.

Release Locks during Remote Calls

Since most distributed deadlocks occur because of a circular lock acquisition attempt, a combination of “Thou shall not hold any lock while making a remote call” and thread pools with large maximum sizes is a simple and effective strategy to avoid deadlocks.

With Ice, there are two common ways to make remote calls outside locks:

• Just release the lock (or locks) explicitly before the remote call (see Listing 1).
• Send the request asynchronously, as oneway or using AMI (see Listing 2 and Listing 3).

Be careful with oneway and AMI calls: they can block, which may trigger some deadlocks (as illustrated by several examples in the first part of this article).

Listing 1: Explicit Release

```cpp
class Client : public IceUtil::Mutex
{
public:
    void f(RemoteObjPrx proxy)
    {
        // Acquire mutex to serialize access
        // to _myState
        Lock lock(*this);

        // In-parameters for the remote call
        // (these must be deep-copies)
        Param1 param1 = _myState.data1;
        Param2 param2 = _myState.data2;

        // Release mutex
        lock.release();

        // Remote call
        Result r = proxy->op(param1, param2);

        // Reacquire mutex
        lock.acquire();

        // Write result
        _myState.data3 = result;
    }

private:
    State _myState;
};
```

Listing 2: Oneway Call

```cpp
class Client : public IceUtil::Mutex
{
public:
    void f(RemoteObjPrx proxy)
    {
        // Get oneway proxy
        RemoteObjPrx onewayProxy = static_cast<
            RemoteObjPrx::uncheckedCast(
                proxy->ice_oneway()));

        // Acquire mutex to serialize access
        // to _myState
        Lock lock(*this);

        // Oneway call (no result this time)
        // Once the message is written to the
        // client’s TCP/IP buffer, op2 returns
        // and the mutex is released
        onewayProxy->op2(_myState.data1, _myState.data2);
    }

private:
    State _myState;
};
```
Listing 3: AMI Call

class Client : public IceUtil::Mutex
{
public:
    class AMICallback : public AMI_RemoteObj_op
    {
public:
        AMICallback(Client& parent) :
            _parent(parent) {}

        void ice_response(const Result& r)
        {  
            _parent.setResult(r);
        }

        void ice_exception(const Exception& e)
        {  
            cerr << "op raised: " << e << endl;
        }
    
private:
        Client& _parent;
    
};

    void f(RemoteObjPrx proxy)
    {
        // Create AMI callback
        AMI_RemoteObj_opPtr callback =  
            new AMICallback(*this);
        Lock lock(*this);
        proxy->op_async(  
            callback, _myState.data1,  
            _myState.data2);
        // I don’t wait for the result
    }

    void setResult(const Result& r)
    {
        Lock lock(*this);
        _myState.data3 = r;
    }

private:
    State _myState;
};

The major downside of this strategy is that locks are often acquired
for a good reason, even during remote calls. While you make the
remote call without any lock, another thread may change your
data—handling these new conditions can dramatically increase
the complexity of your code, as illustrated by Listing 4. It’s not
just more lines of code: this lock acquisition/release/re-acquisition
makes race conditions much more likely. And even when you got
it right, verifying your code’s correctness will be much more time-
consuming.

Listing 4: Doing without Locking

class Player : public IceUtil::Mutex
{
public:
    // Implementation with lock held during  
    // remote calls; straightforward!
    void joinNoRelease(TeamPrx newTeam)
    {
        Lock lock(*this);
        if(_myTeam != 0)
            _myTeam->removePlayer(_myId);
        newTeam->addPlayer(_myId);
        _myTeam = newTeam;
    }

    // This time we release the lock during the
    // remote calls
    void join(TeamPrx newTeam)
    {
        TeamPrx oldTeam = 0;
        Lock lock(*this);
        oldTeam = _myTeam;
        _myTeam = 0;
        lock.release();
        if(oldTeam != 0)
            {
                // Let’s assume _myId is immutable,
                // and as a result usable outside the
                // lock protection.
                oldTeam->removePlayer(_myId);
            }
        // Let’s also assume duplicate adds are
        // ignored
        newTeam->addPlayer(_myId);
        lock.acquire();
        // Check that I did not join another team
        // during these remote calls
        if(_myTeam != 0)
            {
                if(!proxyIdentityEqual(_myTeam,
                    newTeam))
                {
                    // Ok, I did, so newTeam is no
                    // longer relevant
                    lock.release();
                    newTeam->removePlayer(_myId);
                }
                // Else, I joined twice the same team
            }
        else
            {
                // Looks like I did not join another
                // team during the remote calls
                _myTeam = newTeam;
            }
    }

private:
    State _myState;
};

AVOIDING DEADLOCKS
Overall, I advise to avoid this strategy: trading deadlocks for race conditions and/or very complicated code is not a good idea.

**Avoiding Deadlocks**

Detect, Kill, Try Again

Another strategy is to accept that deadlocks will occur: the application expects deadlocks from time to time and, when a deadlock occurs, it recovers and continues. Most if not all database systems use this strategy. Listing 5 illustrates deadlock recovery with Freeze (the Ice persistence service written on top of the Berkeley DB database system): we implement last month’s account transfer operation assuming all accounts are on the same server and are stored in the same Freeze database.

Listing 5: Freeze Deadlock Recovery

```java
// Java
public class Account extends _AccountDisp {

  public void transfer(
    int amount, AccountPrx toAccount,
    Current current)
  throws InsuffientFunds {
    Identity targetId =
      toAccount.ice_getIdentity();
    // transfer to self does nothing
    if(current.id.equals(targetId))
      return;

    // Create a new Freeze connection
    Freeze.Connection connection =
      Freeze.Util.createConnection(
        _communicator, _envName);
    try {
      boolean tryAgain = false;
      do {
        if(tryAgain)
          tryAgain = false;
        tryAgain = false;
        tx = connection.beginTransaction();
        int myBalance = accounts.get(
          current.id).intValue();
        try {
          if(myBalance - amount < 0)
            throw new InsuffientFunds();
          tx.rollback();
          tryAgain = true;
          continue;
        } // Same for target
          if(targetBalance + amount < 0)
            throw new InsuffientFunds();
          accounts.fastPut(targetId,
            new Integer(targetBalance +
            amount));
        }
        catch(Freeze.DeadlockException d)
          { // cleanup
            connection.close();
            if(tryAgain)
              tryAgain = false;
            tryAgain = false;
            tx = connection.beginTransaction();
          } // Get accounts. We assume
            AccountDict is a Freeze dictionary,
            Identity to int
            AccountDict accounts =
              new AccountDict(
                connection, _dbName, true);
            // Start a new transaction
            Freeze.Transaction tx =
              connection.beginTransaction();
        }
      } finally {
        // cleanup
        connection.close();
      }
    
    private Communicator _communicator;
    static private String _envName = "banking";
    static private String _dbName = "accounts";
  }

  Thanks to this strategy, database systems can continue to operate regardless of what their clients (applications, interactive sessions, etc.) do.

  Now, is it possible to use this strategy for a distributed Ice application? First, we need to detect deadlocks, preferably before they occur. The simplest detection mechanism is timeouts: Listing 6 shows the account transfer example (now distributed again) with
Avoiding Deadlocks

Timeouts used to detect deadlocks. (Please refer to the “Invocation Timeout” section of the Ice manual to see how to associate timeouts with proxies.)

Listing 6: Using Timeouts to Detect Deadlocks

```java
public class Account extends _AccountDisp {
    synchronized public void deposit(int amount, Current current)
        throws InsufficientFunds {
        if(amount + _balance < 0)
            throw new InsufficientFunds();
        _balance += amount;
    }
    public void transfer(int amount, AccountPrx toAccount, Current current)
        throws InsufficientFunds {
        // transfer to self does nothing
        if(current.id.equals(toAccount.ice_getIdentity()))
            return;
        boolean tryAgain = false;
        do {
            if(tryAgain)
                // Sleep a little bit outside the synchronization
                Thread.sleep(20);
            tryAgain = false;
        } while(tryAgain);
        synchronized(this) {
            // Verify that this account has sufficient funds
            if(_balance - amount < 0)
                throw new InsufficientFunds();
            // First the deposit on the toAccount
            // - if it raises an exception, we don’t touch _balance
            try {
                toAccount.deposit(amount);
            } catch(TimeoutException e) {
                // We consider that a deadlock occurred
                tryAgain = true;
                continue;
            }
            _balance -= amount;
        }
    }
    private int _balance;
}
```

Using timeouts means blocking for the duration of the timeout. Also, when a remote call times out, Ice closes the underlying TCP/IP connection; closing and then reopening the connection adds some overhead. Depending on the frequency of these deadlocks, the performance impact of relying on timeouts may not be acceptable. Another concern is correctness: with the money transfer example, if the timeout is not due to a deadlock, we could end up depositing twice the same amount. The alternative is to use a more advanced deadlock-detection mechanism: like a database system, we can maintain a graph of locks held and requested, and raise an exception when a loop is detected. Listing 7 shows the Account transfer rewritten with such a mechanism.

Listing 7: Using a Distributed Deadlock Detection Service

```java
interface Locker {
    // Check if I can acquire this resource without deadlocking.
    void add(Ice.Identity id) throws DeadlockException();
    void remove(Ice.Identity id);
    void destroy();
}
interface LockerFactory {
    // Create a new locker with one initial lock (not enough to deadlock)
    Locker createLocker(Ice.Identity id);
}
```

// Java
```java
public class Account extends _AccountDisp {
    void transfer(int amount, AccountPrx toAccount, Current current)
        throws InsufficientFunds {
        // transfer to self does nothing
        if(current.id.equals(toAccount.ice_getIdentity()))
            return;
        boolean tryAgain = false;
        do {
            if(tryAgain)
                // Sleep a little bit outside the synchronization
                Thread.sleep(20);
            tryAgain = false;
        } while(tryAgain);
        synchronized(this) {
            // Verify that this account has sufficient funds
            if(_balance - amount < 0)
                throw new InsufficientFunds();
            // First the deposit on the toAccount
            // - if it raises an exception, we don’t touch _balance
            try {
                toAccount.deposit(amount);
            } catch(TimeoutException e) {
                // We consider that a deadlock occurred
                tryAgain = true;
                continue;
            }
            _balance -= amount;
        }
```
// There is a new locker parameter
public void deposit(int amount, LockerPrx locker,
   Current current)
   throws InsufficientFunds, DeadlockException
{
   locker.add(current.id);
   try
   {
      synchronized(this)
      {
         if(amount + _balance < 0)
         {
            throw new InsufficientFunds();
         }
         _balance += amount;
      }
   }
   finally
   {
      locker.remove(current.id);
   }
}

public void transfer(int amount,
   AccountPrx toAccount, Current current)
   throws InsufficientFunds
{
   boolean tryAgain = false;
   do
   {
      if(tryAgain)
      {
         // Sleep a little bit outside the
         // synchronization
         Thread.sleep(20);
         tryAgain = false;
      }
      LockerPrx locker =
         _lockerFactory.createLocker(
            current.id);
      try
      {
         synchronized(this)
         {
            // Verify that this account has
            // sufficient funds
            if(_balance - amount < 0)
            {
               throw new InsufficientFunds();
            }
         }
         toAccount.deposit(
            amount, locker);
      }
      catch(DeadlockException e)
      {
         // Deadlock occurred
         tryAgain = true;
         continue;
      }
      _balance -= amount;
   }
   finally
   {
      // We’re careful to always cleanup the
      // locker
      locker.destroy();
   }
   while(tryAgain);
}

private int _balance;
static private LockerFactoryPrx _lockerFactory = ...
   // Somehow gets a proxy to the factory
   ...

As this last example shows, we had to change the original interface
(to add an extra parameter to deposit), the detection is far from
free (four remote calls per successful transfer), and the code has
become more complicated. A more optimized deadlock detection
mechanism is certainly possible, but regardless of how deadlocks
are detected, the application needs to recover when a deadlock
exception is raised. In the account transfer example, we were
lucky: we did not write anything before the remote call, which
made recovery very simple. However, this is uncommon: often the
recovery will be much more difficult to code, error-prone, and hard
to test (since many deadlocks are not easy to reproduce).

The performance degradation and the increased complexity of
the application code make this strategy far from ideal. Therefore, I
recommend using it only in isolated situations, but not as a general
strategy.

Ordered Call Flows

Another simple deadlock-avoidance strategy is to order calls
between servers: design the application in such a way that calls
between servers flows in only one direction, without any loops.
You need to create a graph using the servers as nodes and the calls
between servers as edges. Then eliminate any loop to defeat the
(circular wait condition (see Part I), which makes deadlocks impossible.
AVOIDING DEADLOCKS

Figure 1 shows the call-flow graph for a simple distributed game server.

When a server calls another server, it must pass along all the data the target will need to perform the requested operation: callbacks, a common source of deadlocks, are forbidden. Eliminating callbacks has also a positive side-effect: the locking issue discussed in the “Thread-Safe Marshaling” article [Connections 2] will mostly disappear since it does not apply to in-parameters.

This strategy has three key advantages over the strategies presented earlier:

• The implementation code remains simple.
• It does not favor race conditions: as long as you follow the documented call flow, you can lock as much as you like.
• It avoids deadlocks with both application locks and Ice thread pools. (With the other strategies, you have to use thread pools with large maximum sizes to avoid thread pool deadlocks.)

Eliminating callbacks will often make caching some data necessary. Let’s take a very simple Player/Team example, shown in Listing 8.

Listing 8: Simple Player/Team example

```c++
// Slice
interface Team;
interface Player
{
    string getName();
    void setName(string newName);
    void join(Team* team);
};
```

interface Team
{
    // Called by Player.join
    void addPlayer(Player* newPlayer);
    void removePlayer(Player* oldPlayer);
    Ice::StringSeq getPlayerNames();
};

With these interfaces, the Team has to call back on the Player proxy to get the player name. A very simple implementation of Team would just maintain a list of Player proxies, add to this list in addPlayer, remove from this list in removePlayer, and call getName on each player proxy in getPlayerNames. Eliminating the callbacks requires some changes to these interfaces; Listing 9 shows the Player/Team without callbacks.

Listing 9: Player/Team without Callbacks

```c++
// Slice
interface Team;
interface Player
{
    string getName();
    void setName(string newName);
    void join(Team* team);
};
```

interface Team
{
    void addPlayer(Player* newPlayer, string playerName);
    void removePlayer(Player* oldPlayer);
    void updatePlayerName(Player* member, string playerName);
    Ice::StringSeq getPlayerNames();
};

Now Team maintains a list of player proxies and caches the player’s name. Since the player name can change, we need to maintain this cache:

Figure 1: Servers and Calls forming an Acyclic Directed Graph
AVOIDING DEADLOCKS

// C++
void PlayerI::setName(
    const string& newName, const Current&)
{
    Lock lock(*this);
    // Update myTeam’s player-name cache;
    // note that I am calling with "this"
    // locked to ensure consistency
    if(_myTeam != 0)
    {
        _myTeam->updatePlayerName(
            _thisProxy, newName);
    }
    _name = newName;
}

You may also consider another interface change: if Team has no operation that returns a Player proxy, you could have your team track players by Ice::Identity instead of by proxy, as shown in Listing 10.

Listing 10: Using Ice::Identity to Prevent Callbacks

// Slice
interface Team;
interface Player
{
    string getName();
    void setName(string newName);
    void join(Team* team);
};

interface Team
{
    void addPlayer(Ice::Identity newPlayerId,
        string playerName);
    void removePlayer(Ice::Identity oldPlayerId);
    void updatePlayerName(Ice::Identity memberId,
        string playerName);
    Ice::StringSeq getPlayerNames();
    // The caller must call clearTeam() on each
    // of these proxies
    PlayerSeq destroy();
};

The advantage is that Team is no longer able to “accidentally” call back on a player (since you need a proxy for a callback); the disadvantage is loss of typing.

You may sometimes face another interesting dilemma: which call-flow direction makes the most sense? With Player and Team, choosing Team→Player instead of Player→Team would make player transfers considerably more complicated, so the answer is obvious. However, when there is no clear-cut choice, I recommend making the calls flow from the server with fine-grained objects to the server with coarse-grained objects: remote calls take time, and, to improve concurrency, it is better to avoid locking coarse-grained objects while making remote calls.

So far, we haven’t seen any problem with this strategy; is it a perfect solution? Unfortunately, it is not always possible or reasonable to make all calls flow in a single direction. For example, if I want to destroy a Team, and release all its current players, what would I do? A not-so-nice solution would have destroy on Team return the proxies to all existing members, and make the caller responsible for notifying each player. Listing 11 shows the reworked interfaces.

Listing 11: Adding destroy while Maintaining the Call Flow

// Slice
interface Team;
interface Player
{
    string getName();
    void setName(string newName);
    void join(Team* team);
    void clearTeam();
};

interface Team
{
    void addPlayer(Player* newPlayer,
        string playerName);
    void removePlayer(Player* oldPlayer);
    void updatePlayerName(Player* member,
        string playerName);
    Ice::StringSeq getPlayerNames();
    // ATTENTION: calls back on players
    void destroy();
};

It would certainly be more convenient for destroy itself to perform this notification, by making calls in the “forbidden” direction; Listing 12 shows this simpler destroy and its implementation.

Listing 12: An Exception to the Ordered Call Flows

// Slice
interface Team
{
    void addPlayer(Player* newPlayer,
        string playerName);
    void removePlayer(Player* oldPlayer);
    void updatePlayerName(Player* member,
        string playerName);
    Ice::StringSeq getPlayerNames();
    // ATTENTION: calls back on players
    void destroy();
};
In other words, the “single direction” can’t be an absolute rule: use it as the general rule and allow a few exceptions. Each exception must be clearly documented, with great care taken in its implementation to avoid deadlocks.

A refinement of this strategy is to look at call flows at a finer granularity: in particular, considering IceBox services one by one often makes more sense than just looking at the IceBox server containing these services. One step further is to use object adapters for the nodes (instead of servers and services), assuming each object adapter has its own thread pool and servants. This can give you more flexibility at the cost of a more complicated graph.

**Conclusion**

If you develop a distributed application without paying any attention to deadlocks and, despite that, manage to avoid them, you are one lucky developer. If you don’t like gambling, you should design your application with a deadlock-avoidance strategy. As we saw, some strategies come with a heavy cost in terms of code complexity and performance and are best avoided. Often, the “ordered call flows” strategy will be your best bet.
FAQ Corner

In each issue of our newsletter, we present a few frequently-asked questions about  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 should I pass proxies?**

We sometimes see Slice definitions such as the following:

```cpp
// Slice
interface ThingManager
{
    string get(/* params */);
    // ...
};

interface Thing
{
    // ...
};

The client code then looks something like this:

```C++
// C++
ThingManagerPrx tm = ...;
string thingStr = tm->get(/* params */);
ObjectPrx obj = communicator->stringToProxy(
    thingStr);
ThingPrx thing = ThingPrx::uncheckedCast(obj);
// Use thing...
```

While this works, it has a number of problems.

For one, returning a string to the client and then converting the string back into a proxy is somewhat inefficient. (Not seriously so, but why waste CPU cycles when there is no need to?)

Second, and more seriously, this design suffers from three flaws:

1. A reader looking at the ThingManager interface in isolation sees that the get operation returns a string. But, as the code demonstrates, what is really returned is a proxy—it’s just that the proxy is returned as a string and converted back to a proxy in the client. In other words, the design obscures the intent, which is that the get operation returns a Thing proxy.

2. The design is vulnerable to programming error. For example, the server could accidentally return a string that is not a stringified proxy. In that case, the client’s call to stringToProxy will throw an exception because it does not parse correctly.

3. The design is not type safe. For example, the server could mistakenly return a stringified proxy to an object other than a Thing. In that case, the client will successfully convert the string into a proxy and, because the client has chosen to use an uncheckedCast, will even be able to down-cast that proxy into a Thing proxy. But of course, when the client makes its first invocation via that proxy, things will go wrong: the client will most likely get an exception but, per chance, an invocation may even work, but do something entirely unexpected.

Replacing the uncheckedCast with a checkedCast detects the problem earlier, but still does not fix the underlying issue, namely, that it is possible for the server to return an object that is-not-a Thing.

Compare the preceding Slice definitions to the following:

```cpp
// Slice
interface Thing
{
    // ...
};

interface ThingManager
{
    Thing* get(/* params */);
};

With this design, the client code looks as follows:

```C++
// C++
ThingManagerPrx tm = ...;
ThingPrx thing = tm->get(/* params */);
// Use thing...
```

For one, this is a lot simpler: the code neither calls stringToProxy, nor does it use a down-cast. Secondly, this design clearly expresses the intent: one look at the Slice definitions tells us that the get operation returns a Thing proxy. And, most importantly, this design is type safe: the server can return only a proxy to an object of type Thing (or a proxy to an object that is derived from Thing), and the client can receive only a proxy of type Thing (or a proxy to a base type of Thing). Passing a proxy of the wrong type in the server causes a compile-time error, as does treating the return value as the wrong type in the client.

Given that the second design is obviously superior, why do we need stringified proxies at all? The answer is that we need them for bootstrapping. The client must have a proxy to at least one object before it can invoke an operation, but to invoke an operation, the client must have a proxy. In other words, stringified proxies solve the chicken-and-egg problem of how to get an initial proxy (or a small number of initial proxies), so a client can start making invocations.

The upshot of all this is that the only time you should use a stringified proxy in your application is during startup. (Most likely, you will get a stringified proxy from a configuration file or as a com-
mand-line argument.) Thereafter, you should never see a stringified proxy again. Instead, pass proxies as a first-class data type just as you would pass integers or structures. This is not only more efficient, but documents the intent of your design much better. And, if you make a mistake in your code, you get to find out at compile time instead of at run time.

**Q:** Why does Ice limit the size of a request?

You may have noticed that Ice limits the size of requests and replies. By default, any received message that is larger than 1MB raises a `MemoryLimitException`. The 1MB limit applies not to the size of the application data (that is, the payload of the request or reply), but to the entire message as sent over the network. This means that the maximum size of the payload is somewhat smaller than 1MB: the Ice protocol requires a few bytes for its protocol header and, in its marshaled form, depending on the data in the message, the payload can be somewhat larger than the sum of the sizes of its data elements. The exact amount of protocol overhead depends on the operation name, the length of the object identity of the target object, and the type of data you are sending; in general, unless you have very small payloads of only a few bytes, the overhead is insignificant.

So, why does Ice apply this arbitrary message size limit? The answer is simple: without a limit, the Ice run time at the receiving end would be at the mercy of the sending side. To understand why, consider what happens inside the Ice run time when data becomes available for reading on a connection: the run time reads a 14-byte protocol header that contains the size of the message, allocates a buffer that can hold the entire message, and then does a single `read` system call to read the remaining size–14 bytes into the buffer. During unmarshaling, the run time reads the message data from this buffer and, once it has unmarshaled the message payload into parameters, deallocates the buffer again.

Consider now what would happen if someone accidentally or maliciously were to send a protocol header that contains a very large message size, such as 1GB. Without a limit on the size of a message, the run time would attempt to allocate a 1GB buffer in preparation for reading the remainder of the message. Assuming that the allocation succeeds, the receiving process will have just increased its virtual address space by a gigabyte. If that process is a server, and a number of clients send such bogus requests, the server will repeatedly run out of memory. At best, all this allocated memory consumes swap space, which negatively affects other processes on the same machine; at worst, the server may simply crash. The message size limitation therefore acts as a safety stop to weed out messages that are “unreasonably large.”

If your application needs to pass more than around 1MB of data with a single RPC, you can change the default limit by setting the value of the `MessageSizeMax` property. This property contains the maximum message size in kilobytes. Note that, if your application only exchanges small amounts of data, it does no harm to set `MessageSizeMax` to an appropriate smaller value. Doing so ensures that any bogus messages are rejected as soon as the protocol header is read.

For UDP messages, Ice uses two additional properties, `UDP.SndSize` and `UDP.RcvSize`. (Note that the value of these properties is in bytes, not kilobytes.) These properties determine the maximum size for the sending and receipt of UDP messages. The default value depends on your operating system. (Typical defaults are 64kB on Windows, and 8kB on Linux.) The maximum sensible value for these properties is 65535 (64kB). You can set them to larger values, but doing so merely increases the size of the kernel buffers and does not affect the hard limit of 65507 bytes for the payload of a UDP packet. (It follows that, for UDP, the maximum Ice message size, including protocol overhead, is 65507 bytes because that is the limit imposed by UDP.) If you set `MessageSizeMax` in addition to the UDP send/receive size, the smaller of the property settings applies.

Finally, Ice not only limits the overall size of requests but, during unmarshaling, also provides a number of sanity checks on the received data. For example, the run time keeps track of how much data has been unmarshaled so far and, whenever it unmarshals a sequence, it ensures that the sequence can fit into remaining buffer space for the yet-to-be-unmarshaled portion of the message. This makes it impossible for a malicious sender to craft messages with bogus sequence sizes that would result in memory allocations whose sum of sizes would exceed the limit set by `MessageSizeMax`. 