Connections

ZeroC's Newsletter for the Ice Community

Issue Number 6, September 2005

Caveat Emptor

The hot standard at the OMG these days is the Data Distribution Service, or DDS. DDS provides a simple publish-subscribe mechanism with a real-time focus; as such, it is not really comparable to a distributed object infrastructure such as Ice or CORBA. DDS grew out of NDDS, a product from Real Time Innovations. RTI's defense customers like products that implement open standards, so RTI crafted a standard for NDDS within the OMG. Because several companies have announced that they will implement it, this standard is already considered a success—a rare feat for recent OMG specifications. But let's look beyond the hype: from a user's point of view, standardization suggests many benefits, such as the ability to easily port to another implementation, interoperability between different implementations, and a long lifespan for the technology. Does DDS deliver on these?

DDS defines a portable API using CORBA IDL: even though DDS itself has nothing to do with CORBA or distributed objects, if you want to write portable code, you'll have to familiarize yourself with the CORBA language mappings. This is not ideal, given that CORBA is widely considered a legacy standard, and its language mappings show their age: for example, the C++ mapping, conceived before the standardization of C++ in 1998, has its own private types (such as for strings and sequences) and overly complicated memory management rules. It's hard to see how the standardization committee could have avoided this dependency; if you want an OMG standard, some CORBA baggage is expected. Unfortunately, in addition to this CORBA dependency, the DDS API itself is quite complex and arcane; for example, just like C, the API reports errors with error numbers rather than exceptions. The proprietary API of existing DDS implementations is simpler; RTI still highlights the simplicity of NDDS programming over CORBA: "*...a simple program based on CORBA takes 659 lines of code to NDDS's 107*"). So if you use NDDS, you have to choose between a complex portable API and a simple non-portable one—not an easy trade-off.

Even if the portability of DDS comes at a price, at least it is interoperable—or is it? This recent press release provides some insight: work on DDS interoperability is *about to start*. At best, this means a specification in two or three years, with the hope that some vendors eventually will make their products interoperable; at worst, it could mean a specification in five years (or no specification at all), or a specification without implementations because, by the time the standard is finalized, vendors have lost interest and moved on to greener pastures.

So why bother creating a standard for DDS? Pushing a specification through the OMG is a difficult and long process; why do this just for a new complicated API? OMG standards are created by vendors; in this case, RTI, Thales, and OIS. The main return on investment for their efforts is the new "standard-compliant" check-box on their existing products (NDDS for RTI, SPLICE for Thales). Even better, these companies can now lobby to make the DDS standard—and their respective products—a mandatory choice in some defense programs. It's ironic to see standardization being used to curtail competition and serve the interests of a few vendors (who advertise "no vendor lock-in") while, at the same time, customers end up with an API that makes it harder for them to write their software. *Caveat emptor*: before you commit to a standard, ask yourself whom that decision will serve—it may not be you.

Bernard Normier
Senior Software Engineer
ZeroC, Inc.

Issue Features

Advanced IcePack

In this installment of the series of articles on the chat application, Matthew Newhook shows you how to use IcePack to deploy the application over multiple servers and how to dynamically scale the application as user demand increases.

Taxing Times

Excel is a popular and powerful calculation tool, and you don't have to be a programmer to use it effectively. In this article, Michi Henning shows you how you can seamlessly integrate Ice and Excel, and allow non-programmers to provide the bulk of application-specific code.

Contents

Advanced IcePack

Matthew Newhook, Senior Software Engineer

Introduction

In the previous article, we introduced IcePack and demonstrated how it can simplify the development and deployment of a complex application. This article will continue to focus on IcePack.

Recap

Let us recap the various Ice Objects in our application so far.

- `ChatSessionManager`. Used by Glacier2 to create new `ChatSession` objects.
- `PermissionsVerifier`. Used by Glacier2 for validating user name and password.
- `ChatSession`. Entry point to the chat service from chat clients. Using this object, clients subscribe to new chat rooms. It also manages the life cycle of current subscriptions to chat rooms.
- `ChatRoomParticipant`. Interface for chat clients to send new messages to a chat room.
- `ChatRoomObserver`. Interface for chat clients to receive new messages in a chat room.
- `InvitationCallback`. Interface for chat clients to receive invitations into a chat room.
- `IceStorm`. Responsible for distributing events from the chat room publishers to the chat room subscribers.

In addition, the chat server itself uses the following non-Ice objects:

- `UserManager`. Provides a mapping of user id to `InvitationCallback` proxies.
- `RoomManager`. Repository of chat rooms. This maps the name of the chat room to the chat room topic proxy. Manages the life cycle of a chat room.

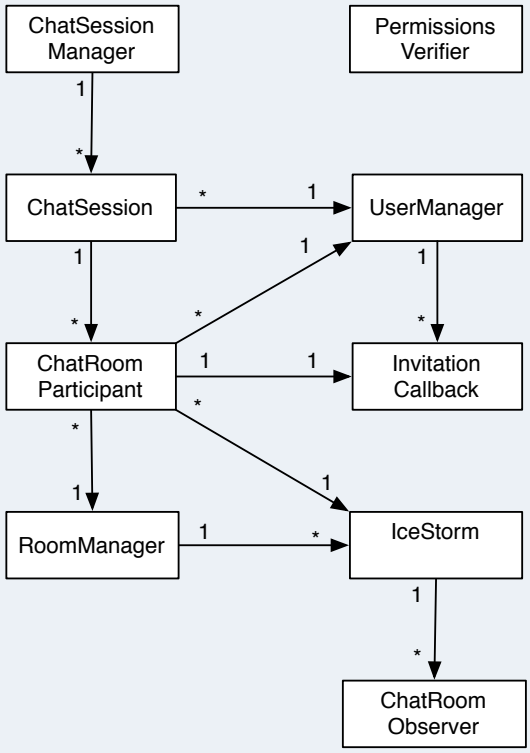
Figure 1 shows a class diagram of the application.

Figure 2 shows the deployment of the server back end.

Deployment Issues

The deployment described in the previous article is not all that flexible. In particular, it is currently impossible to have multiple deployments of any of the servers, but it is reasonable to expect that, once the number of users logged into the system becomes large, we will need multiple Glacier2 routers, multiple chat session servers, and multiple IceStorm servers. (Figure 3 depicts a more scalable deployment.)

Figure 1: Class Diagram



So, what prevents us from deploying multiple chat session servers? Apart from some configuration issues, the major obstacle is the way we have used IceStorm. Consider what would happen if we were to deploy multiple chat session and IceStorm servers with the current setup: each server would act as an island, that is, a chat room "*global*" in one server would be separate from a chat room "*global*" in another server. In order to hook the chat rooms up, they both would need to use the same IceStorm instance. However, the

Figure 2: Server Deployment

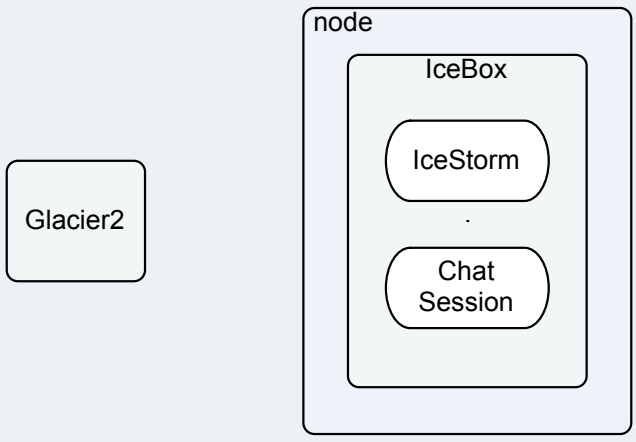
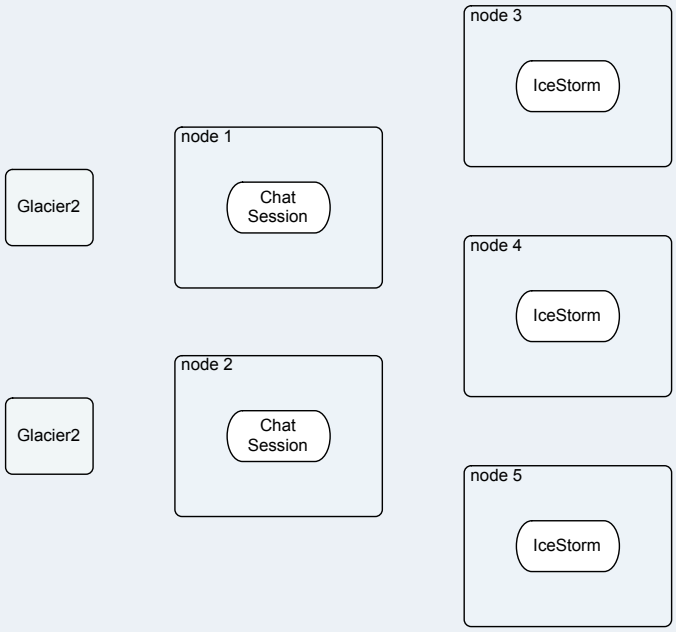


Figure 3: More Scalable Deployment



major impediment to this is the `RoomManager`, which expects to exclusively manage the IceStorm instance. If two `RoomManager` objects manage the same IceStorm instance, the current implementation of the `Topic` life cycle management breaks down. To make matters worse, it's very hard to change the `RoomManager` implementation to work in this way.

Fortunately, there is an easier solution: make the `RoomManager` a separate service. The `RoomManager` only acts as an intermediary between the `ChatSession` and IceStorm when the user enters and leaves a chat room. This is not a frequent operation in comparison to the actual act of chatting and therefore not performance-sensitive: a single `RoomManager` should be able to manage a large number of users.

Perceptive readers will also notice that the server has the same problem with the `UserManager`. If the `UserManager` does not become its own service, users using one chat session server cannot invite users using a different chat session server. For the purposes of this article, we will ignore this issue and return to it in the next installment.

Room Manager Service

The implementation of the `RoomManager` is quite straightforward. Essentially, we're taking the existing implementation and turning it directly into an Ice object. First, the interface:

```
// Slice
interface RoomManager
{
    ChatRoomObserver* enter(string room,
        ChatRoomObserver* observer);
    void leave(string room,
        ChatRoomObserver* observer);
};
```

Now we take the implementation of the `RoomManager` and make it into a servant. First, we rename it to `RoomManagerI`, and add `Current` to the end of each of the methods as follows:

```
// C++
class RoomManagerI : public RoomManager,
                     public IceUtil::Mutex
{
public:
    RoomManagerI(
        const IceStorm::TopicManagerPrx&);
    ~RoomManagerI();

    virtual ChatRoomObserverPrx enter(
        const string&, const ChatRoomObserverPrx&,
        const Current&);
    virtual void leave(
        const string&, const ChatRoomObserverPrx&,
        const Current&);

private:

    // ...

};
```

The implementation of this object has not changed, except each method has the `Current` argument. The next step is to create an IceBox service for the `RoomManager` object, which follows the same pattern as the implementation for the `ChatSessionService`. The implementation is as follows:

```
// C++
class RoomManagerServiceI :
    public ::IceBox::Service
{
public:
    virtual void start(
        const string& name,
        const CommunicatorPtr& c,
        const StringSeq& args)
    {
        TopicManagerPrx tm =
            TopicManagerPrx::checkedCast(
                c->stringToProxy(
                    "IceStorm/TopicManager"));
        _adapter = c->createObjectAdapter(
            "RoomManager");
        _adapter->add(new RoomManagerI(tm),
            stringToIdentity("RoomManager"));
        _adapter->activate();
    }
```

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

private:
    ObjectAdapterPtr _adapter;
};
```

Next, we convert the chat session server to use a room manager proxy. First, we must retrieve the proxy. Since IcePack manages the `RoomManager` object, we can retrieve the proxy via a well-known identity. Note that, by virtue of this change, the `ChatSessionService` neither uses nor is aware of IceStorm. The `ChatSessionServiceI::start` method becomes:

```
// C++_
void
ChatSessionServiceI::start(
    const string& name,
    const CommunicatorPtr& c,
    const StringSeq& args)
{
    _adapter = c->createObjectAdapter(
        "ChatServer");
    RoomManagerPrx rm =
        RoomManagerPrx::checkedCast(
            c->stringToProxy("RoomManager"));
    _adapter->add(
        new DummyPermissionsVerifierI,
        stringToIdentity("verifier"));
    _adapter->add(new ChatSessionRmI(rm),
        stringToIdentity("ChatSessionRm"));
    _adapter->activate();
}
```

Next, we remove the local `RoomManager` implementation from the `ChatSessionService`, and change the remaining references from `RoomManagerPtr` to `RoomManagerPrx`.

The last step is to add the room manager service to the application deployment. We'll add a deployment descriptor called `roommanager.xml`, as follows:

```
// XML
<icepack>
  <service name="${name}"
   entry="RoomManagerService:create">
    <adapters>
      <adapter name="RoomManager" endpoints="tcp">
        <object identity="RoomManager"
         type="::Chat::RoomManager"/>
      </adapter>
    </adapters>
  </service>
</icepack>
```

Our deployment now has three services: IceStorm, chat session, and the room manager. The IceStorm service descriptor, `icestorm.xml`, looks like this:

```
// XML
<icepack>
  <service name="${name}"
   entry="IceStormService,21:create">
    <dbenv name="${service}"/>
      <adapters>
        <adapter name="${service}.TopicManager"
         endpoints="tcp">
        <object identity="${service}/TopicManager"
         type="::IceStorm::TopicManager"/>
      </adapter>
      <adapter name="${service}.Publish"
       endpoints="tcp"/>
    </adapters>
  </service>
</icepack>
```

The chat session service descriptor, `chatsession.xml`, look as follows.

```
// 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>
  </service>
</icepack>
```

Note that we changed the identity of the `ChatSessionManager` and `PermissionsVerifier` object. Why do this? The answer is simple: because we want to deploy multiple chat session services and because distinct objects must have unique identities, we must make the identities of these objects different for each service.

It is clear that we need multiple `ChatSessionManager` objects. Do we really need multiple `PermissionsVerifier` objects as well? We first must decide whether each `PermissionsVerifier` is actually unique. Do they all have the same state? The answer is yes; otherwise a user would have different passwords when they try to log on to a separate front end. This means that, if there are multiple `PermissionVerifier` objects, they are actually replicas of one another—that is, they are conceptually instances of the same object. Of course, the purpose of replicas is fault tolerance: if one replica dies, the system can continue to function. However, the cost is that the state of replicas must be shared, which can be quite complex. (You might think that a simple way to share the state would be to put the information in a central database and have the different instances access

this state. However, you now must ensure that the database itself doesn't become a single of point of failure.) For now, we'll ignore this problem and simply deploy multiple `PermissionVerifier` objects.

Note that, because the identity of the session manager object and the permissions verifier object has changed, the Glacier2 configuration has to reflect this change.

Next, we'll create a single node deployment, `single_node.xml`. This deployment is convenient for testing since the entire deployment is on one node:

```
// XML
<icepack>
  <application name="ChatServer">
    <node name="node">
      <server name="ChatApplication"
       kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="IceStorm"
         descriptor="icestorm.xml"/>
        <include name="RoomManager"
         descriptor="roommanager.xml"/>
        <include name="ChatSession"
         descriptor="chatsession.xml"/>
      </server>
    </node>
  </application>
</icepack>
```

A Better Room Manager

Is this setup that much better? With the deployment descriptor as outlined above, not really. However, the beauty of this setup is that we can change the deployment to add more IceStorm servers and more chat session services without having to change the code.

Or can we? It turns out that, actually, we cannot do this yet because the `RoomManager` implementation uses only one IceStorm instance (namely the one with the Identity `IceStorm/TopicManager`). Let's remedy that.

Consider what the `RoomManager` object really needs to do. At the time of channel creation, it needs to locate an IceStorm instance to use and create a new `Topic` on that instance. Does the `RoomManager` care which instance of IceStorm it's using to create a new `Topic`? The answer is no; each IceStorm instance to the `RoomManager` is equivalent and each provides the same service so, from a service point of view, it doesn't matter which IceStorm instance the `RoomManager` uses.

How about load balancing? Would it be a good idea to keep a list of channels created on each instance of an IceStorm service and try to keep the number of channels on each roughly equivalent? The short answer is that this really is not a good idea: as it turns out, in most cases, a random selection scheme is just as good as more sophisticated schemes (and random selection is certainly much simpler).

So, at channel creation time, the `RoomManager` needs to pick an instance of an IceStorm service at random. What's the easiest way to do that? Let's review the `IcePack::Query` interface. (You can find full documentation for the interface in the Ice distribution.)

```
// Slice
module IcePack
{
interface Query
{
    nonmutating Object*
    findObjectById(Identity id)
        throws ObjectNotExistException;
    nonmutating Object*
    findObjectByType(string type)
        throws ObjectNotExistException;
    nonmutating ObjectProxySeq
    findAllObjectsWithType(string type)
        throws ObjectNotExistException;
};
};
```

There are two sets of operations. `findObjectById` locates objects by identity, and the other two methods locate objects by type:

- `findObjectByType` returns a random object from all objects that support the given type.
- `findAllObjectsWithType` returns all objects that support the given type.

The `RoomManager` could call `findAllObjectsWithType` upon start-up, cache the return value, and then pick an IceStorm object at random from the cached list. This has the advantage that the `RoomManager::enter` method would not need to call on IcePack each time a user enters a chat room. However, it has a serious disadvantage: with this implementation, it is no longer possible to add new IceStorm services at runtime without restarting the `RoomManager`, but the `RoomManager` currently does not support restart.

On the other hand, `findObjectByType` fits the bill nicely: when it is time to create a new `Topic` for a chat room, this method can be called to pick an instance at random from all the available instances.

Let's run through the changes necessary to do this. First, we need to get a proxy to the IcePack `Query` interface:

```
// C++
IcePack::QueryPrx query =
    IcePack::QueryPrx::checkedCast(
    communicator->stringToProxy("IcePack/Query"));
```

We'll store this value in the member variable `_query` of the `RoomManager` implementation. When it's time to create a new chat room, we'll use IcePack to find an instance of `TopicManager` at run time.

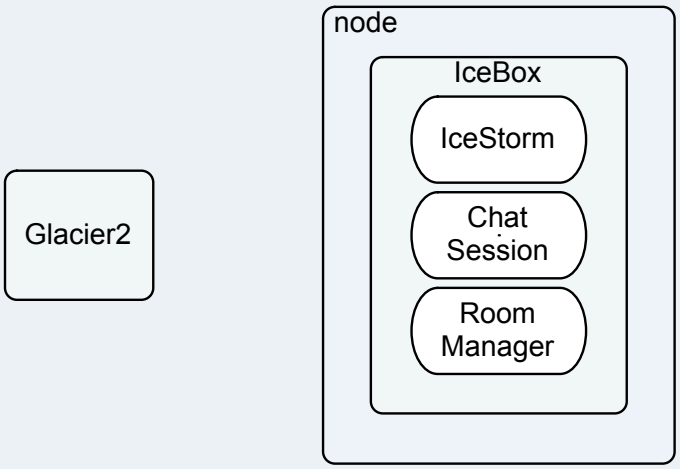
```
// C++
IceStorm::TopicManagerPrx manager =
    TopicManagerPrx::uncheckedCast(
        _query->findObjectByType(
            "::IceStorm::TopicManager"));
topic = manager->create(room);
```

With this approach, we have much more flexibility: we can deploy any number of IceStorm services and the `RoomManager` uses them to create new chat rooms. Furthermore, we can deploy more IceStorm services at run time, and the `RoomManager` will start to use these without a restart!

Changing Deployments

The previous deployment was on a single node, with all services provided within a single physical server, as shown in Figure 4.

Figure 4: Deployment in a Single Server



For testing and development purposes, this is convenient. However, this particular deployment can also cause problems.

Start-up Ordering Problems

Note that the ordering of the services in `single_node.xml` descriptor is very important. The services must be ordered as shown because of their start-up dependencies.

In response to the call to `ChatSessionServiceI::start`, the implementation makes Ice-mediated invocations upon the room manager service in form of calls to `checkedCast`, which make remote call to verify that the Ice object supports the given interface.

```
// C++
RoomManagerPrx manager =
    RoomManagerPrx::checkedCast(
      communicator->stringToProxy("RoomManager"));
```

It follows that IcePack must deploy the services in a particular order. Note that this is necessary only because the services are in the same IceBox server, and because the Ice-mediated invocations are done within the context of the `Service::start` method. If the services were deployed in separate servers, or the calls were done after `Service::start` had completed, this wouldn't be necessary. (Frequently, you can avoid making invocations in the `Service::start` method. For example, you could use an `uncheckedCast` instead of a `checkedCast` to avoid the Ice-mediated call, at the expense of error checking.)

Instability

If the services are all deployed in a single server, all services die if one of them crashes. With our application in its present state, this is no big deal because the restart time is short. However, for more sophisticated applications, the start-up time could be significant.

On the other hand, if services are deployed over multiple physical servers, then a crash won't take down the whole application. Of course, for this to work, the application must be able to recover from the failure of a component gracefully. (Also note that a failure doesn't necessarily indicate a programming bug. Hardware failures can and do occur!)

Can the current implementation recover from a failure? That's a good question and the short answer is no. We will discuss the reasons for this in a future article.

Ability to Restart

A common scenario is that you've discovered a bug, or added a new feature, and want to test your changes. Quite often, this means that you want to restart a service, but not necessarily the entire deployment. If all services are located within one physical server and you are using C++, this is typically not a problem because shared libraries can be unloaded and reloaded at run time. However, if you are using Java, this is a real problem because you cannot unload a `.jar` file: if you want to restart a Java service, you must restart the entire server.

Separate Service Deployment

Fortunately, it is easy to create a deployment that puts each service in its own server:

```
// XML
<icepack>
  <application name="ChatServer">
    <node name="node">
      <server name="IceStorm" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="${server}"
         descriptor="icestorm.xml"/>
      </server>
```

```
      <server name="RoomManager" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="${server}"
         descriptor="roommanager.xml"/>
      </server>
      <server name="ChatSession" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="${server}"
         descriptor="chatsession.xml"/>
      </server>
    </node>
  </application>
</icepack>
```

This configuration uses `${server}` as the service name. (The point of this will become apparent shortly.) If you need to restart a service after a bug fix, such as the chat session service, you can stop and start the service selectively:

```
$ icepackadmin.exe --Ice.Config=config.icepack -e
"server stop ChatSession"

$ icepackadmin.exe --Ice.Config=config.icepack -e
"server state ChatSession"
```

The second command should report the state as `inactive`. When you re-run the chat client, IcePack will restart the chat session service on demand, and you can test your bug fix!

Real-World Deployment

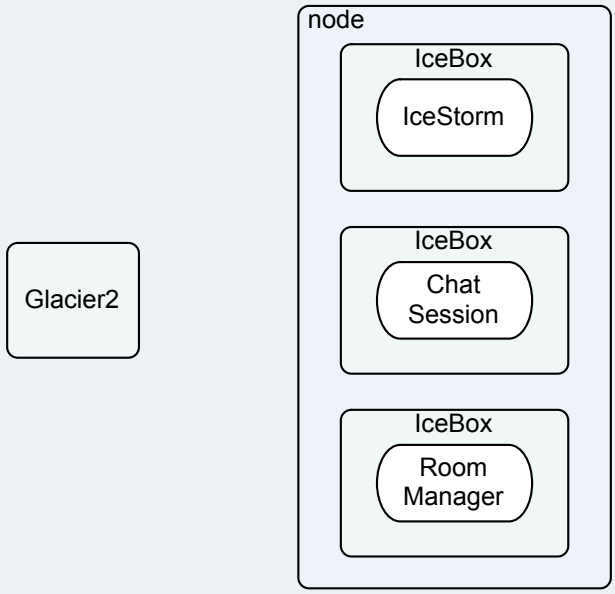
The preceding deployment descriptor is also more suitable for a real-world deployment. If you refer to Figure 1 in the document we'll need to deploy additional Glacier2 routers, chat session services and additional IceStorm services as the need arises. For each chat session service we will need an additional Glacier2 router. (We will only ever have one room manager.) Let's try adding an additional chat session service.

First, we clean out the IcePack node databases. We start `icepacknode`, deploy `single_node_separate.xml`, start up Glacier2, and run the chat client to test the deployment.

Now, to deploy an additional chat session service, we alter the deployment descriptor as follows and save it as a new file, `single_node_separate2.xml`.

```
// XML
<icepack>
  <application name="ChatServer">
    <node name="node">
      <server name="IceStorm" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="IceStorm"
         descriptor="icestorm.xml"/>
      </server>
      <server name="RoomManager" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="RoomManager"
         descriptor="roommanager.xml"/>
      </server>
      <server name="ChatSession" kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="${server}"
         descriptor="chatsession.xml"/>
      </server>
      <server name="ChatSession-2"
       kind="cpp-icebox"
       endpoints="tcp -h 127.0.0.1"
       activation="on-demand">
        <include name="${server}"
         descriptor="chatsession.xml"/>
      </server>
    </node>
  </application>
</icepack>
```

Figure 5: Separate Deployment



The use of `${server}` means that only the name of the chat session service needs to be changed. Now we can update the application.

```
$ icepackadmin.exe --Ice.Config=config.icepack -e
"application update 'single_node_separate2.xml'"
```

Next, we must change the session manager and permissions verifier proxies that the Glacier2 router uses. We can write a new configuration file for each Glacier2 router instance, or we can pass the proxies to the permissions verifier and session manager on the command line. For ease of maintenance we'll pass them on the command line.

We start an additional Glacier2 router for the newly-deployed `ChatSession-2`. Since we are running them on the same host, we'll need to use a different port for the second Glacier2 router. The configuration file properties are overridden on the command line:

```
$ glacier2router --Ice.Config=config.glacier2 --
Glacier2.SessionManager=ChatSession-2-ChatSessionM
anager --Glacier2.PermissionsVerifier=ChatSession-
2-verifier --Glacier2.Client.Endpoints="ssl -p 100
06"
```

Now this new Glacier2 instance needs to be provided to the client. For this demo, we'll modify the properties on the command line as we did with the Glacier2 router.

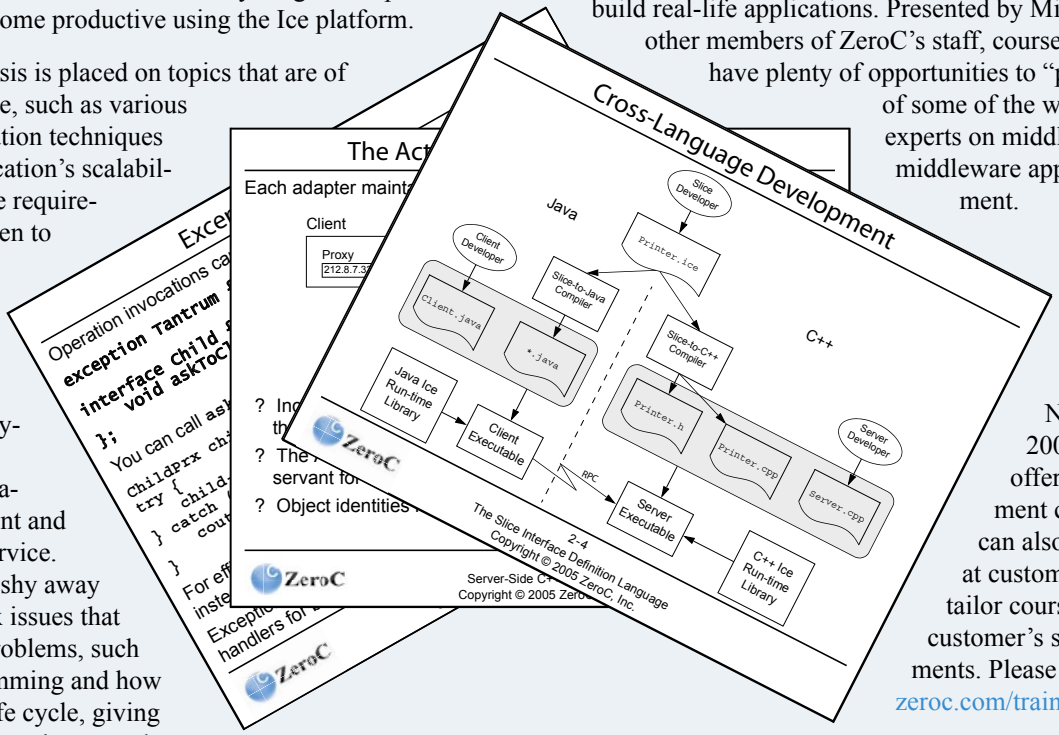
```
$ client --Ice.Default.Router="Glacier2/router:ssl
-p 10005:ssl -p 10006" --Chat.Client.Router="Glaci
er2/router:ssl -p 10005:ssl -p 10006"
```

There we have it! We have deployed a second Glacier2 router and chat session service. Clients can use the new service immediately, without a restart of the chat server!

New Ice Training Courses get Developers up to Speed

ZeroC is in the process of developing two new training courses to help programmers get up to speed on developing Ice applications. The courses (one using C++, the other one using Java) are developed by middleware training expert Michi Henning. Apart from covering the basics, such as the Slice language and the C++/Java language mapping, the courses cover everything developers need to quickly become productive using the Ice platform.

Particular emphasis is placed on topics that are of real-world relevance, such as various servant implementation techniques to achieve an application's scalability and performance requirements, how and when to use asynchronous invocation and dispatch, and how to use parts of Ice that are important for deployment, such as Ice's configuration mechanism and deployment and server activation service. The courses do not shy away from more complex issues that commonly cause problems, such as threaded programming and how to manage object life cycle, giving developers an opportunity to explore application development as it relates to building real applications, as opposed to toy examples.



Each training course extends over three days and includes several practical programming exercises to help attendees become familiar with Ice, and gain first-hand experience of how to build real-life applications. Presented by Michi Henning and other members of ZeroC's staff, course attendees also have plenty of opportunities to "pick the brains" of some of the world's foremost experts on middleware design and middleware application development.

The first public-enrolment course (using C++) will take place in Florida, November 16-18, 2005. Apart from offering public-enrolment courses, ZeroC can also provide training at customer premises and tailor course content to meet customer's specific requirements. Please visit http://www.zeroc.com/training/ for details.

Taxing Times

Michi Henning, Chief Scientist

In Australia (which is where I live), it is yet again the time where people have to submit their annual tax return. (The Australian financial year runs from July to June.) And, this year, income tax rates have changed, so the handy income tax calculator I wrote a few years ago is out of date. Well, no problem, I'll just write another one. And, of course, in keeping with my day-to-day activities, the calculator uses Ice.

The interface to the tax calculator couldn't be simpler:

```
// Slice
module Tax
{
    interface Calculator
    {
        double calculateTax(double income);
    };
};
```

Below is an example session showing the interactions of a simple command-line client with the server.

```
$ ./client
Enter income (0 to quit): 78500
Tax is: 21270
Enter income (0 to quit): 98000
Tax is: 29610
Enter income (0 to quit): 0
$
```

As you can see, someone earning $78,000 pays $21,270 in tax, whereas someone earning $98,000 pays $29,610 in tax.

"So what?" you may ask—after all, this isn't exactly exciting. In fact, the client code is so trivial that I won't even bother showing it. (Although, as usual, you can download the code for this article from our web site.) And, the server code, apart from the tax calculation itself, is not very exciting either, so what's the big deal?

Integrating the Calculator with Excel

Well, having been left with an outdated tax calculator once, I decided that writing tax calculations in C++ (or any other programming language for that matter) is somewhat archaic. It would be nicer to have the tax calculation outside the server code so, next time there is a change in tax rates, it won't be necessary to modify the source code and redeploy the server. Moreover, Microsoft Excel is a popular tool for such calculations, so why not use it? The next time tax rates change, my accountant can send me an updated spreadsheet and, to get my server to use the latest tax rates, all I'll have to do is save the spreadsheet. Figure 1 shows this architecture.

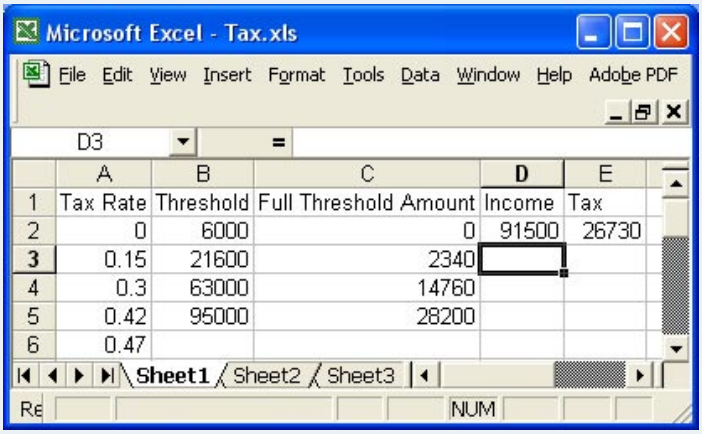
Of course, my tax example is somewhat contrived, but the approach is attractive: by using Excel, we allow a non-programmer with domain-specific knowledge (such as an accountant) to implement or customize the application logic; an Ice server then makes the data available to remote clients, but delegates the actual calculation to the spreadsheet. If there is a change to some aspect of the calculations, only the spreadsheet need change, not the server, and no programmer need be involved. And, of course, this idea can be

Figure 1: Delegating Calculations to Excel

applied to any number of other business calculations; an obvious example would be inventory and pricing information for an online web store for which we definitely do not want to call a programmer every time new stock arrives or the price of an item changes.

Figure 2 shows a screen shot of the Excel spreadsheet that implements the tax calculation.

Figure 2: Server Deployment



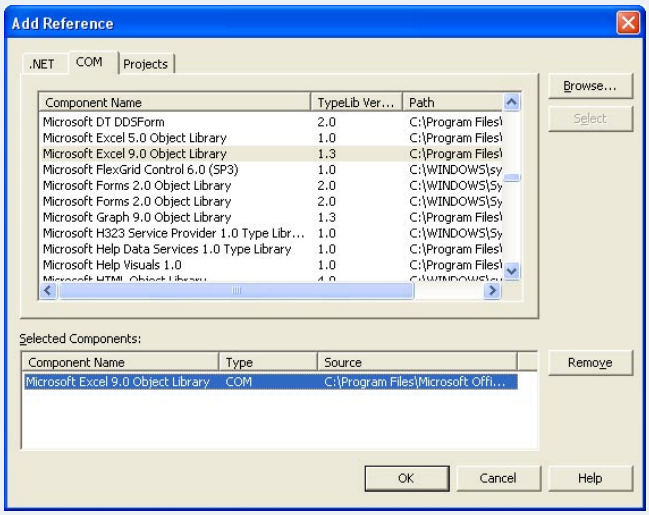
Cell D2 is an input field where you enter the income, and cell E2 contains a formula that calculates the tax due on that income and displays it. (The remaining cells contain the tax rates for various income thresholds and are used in the formula that is attached to E2.)

Given this spreadsheet, how can we access it from an Ice server? As it turns out, Excel comes with a COM library that allows you to drive the application via "remote control." The library allows a program to do anything that you could do interactively, which is just what we need.

Figure 3 shows the dialog that adds a reference to the Excel COM library to a Visual Basic project. (I have written the code that follows in Visual Basic, mainly because that is a popular language for application integration; however, you can write the server in whatever language you prefer.)

Note that, on my machine, there are two versions of the library (5.0 and 9.0) because I have two versions of Excel installed. (The version on your machine may differ from either of these—simply use the highest-numbered version if you have more than one.) Having added a reference to the library to the Visual Basic project (as well as a reference to the Ice run time, of course), we can be on our merry way. Here is the bulk of the server code:

Figure 3: Adding a Reference to the Excel COM Library




```
Imports System

Module ExcelServer
    Private Class TaxCalculator
        Inherits Tax._CalculatorDisp

        Public Sub New( _
               ByVal worksheet As Excel.Worksheet)
            Me.worksheet = worksheet
        End Sub

        ' More code here...

        Private worksheet as Excel.Worksheet
    End Class

    Sub Main(ByVal args As String())
        Dim status As Integer = 0
        Dim xl As Excel.Application = _
            Microsoft.VisualBasic. _
                CreateObject("Excel.Application")
        xl.Visible = True
        xl.UserControl = True
        Dim wb As Excel.Workbook
        Dim file As String = _
            IO.Directory.GetCurrentDirectory() _
                & "\Tax.xls"
        Try
            wb = xl.Workbooks.Open(file)
            Dim app As Calculator = _
                New TaxCalculator(wb.ActiveSheet)
            status = app.main(args)
        Catch ex As Exception
            Console.Error.WriteLine(ex)
            status = 1
        Finally
            If Not wb Is Nothing Then
                wb.Close(False)
            End If
            xl.Quit()
        End Try
        Environment.Exit(status)
    End Sub
End Module
```

The `TaxCalculator` class is the servant class that implements the `calculateTax` operation. For the moment, the `calculateTax` method is omitted—we'll get to that shortly. For now, notice that the class has a private member `worksheet` of type `Excel.Worksheet` (initialized by the constructor), which we use to interact with the spreadsheet as the client calls `calculateTax`.

If you have a look at `Main`, you will see that the program instantiates an object `xl` of type `Excel.Application`. This is the main handle that is used by the program to interact with a running instance of Excel. The code sets the `Visible` and `UserControl` properties of the application to `True`, so Excel is visible and allows a user to interactively manipulate the contents of the spreadsheet while the server is running. (You can, for example, change the formulas or tax thresholds while the server is running and the client gets to see the effects of your changes as soon as it calls `calculateTax`.)

The code then opens a spreadsheet called `Tax.xls` (which is expected to be in the server's current directory) and instantiates the `TaxCalculator` servant class, passing the active worksheet to the constructor. Once the server shuts down, it closes the spreadsheet without saving the contents and quits Excel.

What remains to be shown is the code for the implementation of `calculateTax`, which is very simple:

```
Public Overloads Overrides Function _
    calculateTax(ByVal income As Double, _
                 ByVal current As Ice.Current) _
                 As Double
    worksheet.Cells(2, 4) = _
       income worksheet.Calculate()
    Return Double.Parse( _
              worksheet.Cells(2, 5).Value)
End Function
```

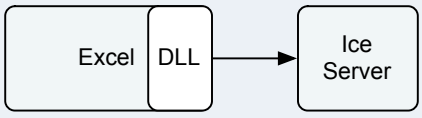
As you can see, the operation simply inserts the income into the fourth cell in the second row, calls `Calculate` on the worksheet, and returns the contents of the fifth cell in the second row as a `Double` value. As shown, the application is not particularly robust and needs obvious improvements such as better error handling and configuration. However, it illustrates that you can easily integrate the functionality provided by Excel with your Ice applications.

Excel as an Ice Client

Delegating calculations to Excel in this manner is useful, but you may have noticed that there is nothing all that Ice-specific to this example—it just so happens that the application that controls Excel is also an Ice server. A more interesting use case is to have Excel act as an Ice client, for example, to populate an Excel spreadsheet with data that is provided by an Ice server. "No problem," you may think. After all, Excel includes Visual Basic for Applications (VBA), so we could use that to make Excel an Ice client. Unfortunately, this does not work: VBA has a number of quite drastic limitations (such as the inability to call constructors that require arguments), which make it impossible to call into the Ice run time (or into Slice-generated code) from VBA.

Fortunately, there is a way out: if you have the Visual Studio Tools for Office (sold as an add-on product for Visual Studio), you can create.NET code that is callable directly from within Excel. This method does not suffer the limitations of VBA, which means that you can interact with an Ice server directly from within Excel. (You need Office 2003 or later for this.). Figure 4 shows the system architecture using Excel as an Ice client.

Figure 4: Using Excel as an Ice Client



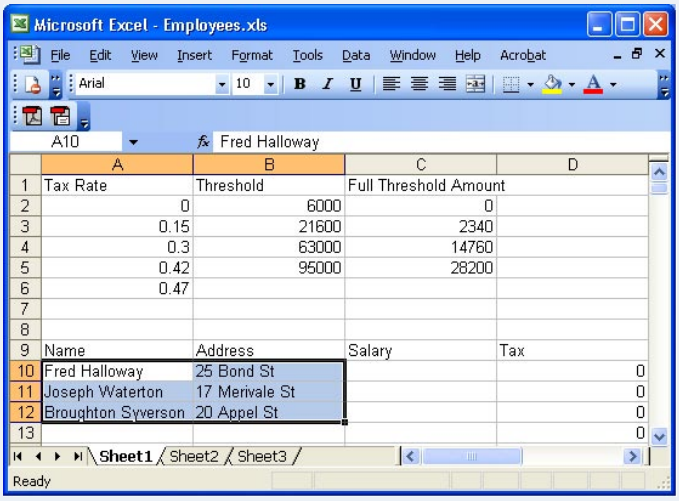
To illustrate how all this works, let's stay with the tax example. Suppose we have a number of employees and their address details in a business database. The database is remotely accessible via an Ice server. At the end of the tax year, we need to produce a report that shows each employee's name, address, salary earned for that year, and the tax payable on that salary. The Slice definitions for this application are as follows:

```
module Employees
{
    struct EmployeeDetails
    {
        string name;
        string address;
        // ...
    };

    interface Employee
    {
        EmployeeDetails getDetails();
        void setAnnualSalary(double salary);
        void setTaxPayable(double amount);
        // ...
    };

    ["vb:collection"]
    sequence<Employee*> EmployeeSeq;

    interface EmployeeFinder
    {
        EmployeeSeq list();
        // ...
    };
};
```

The `list` operation of the `EmployeeFinder` interface returns a sequence of `Employee` proxies; the `getDetails` operation returns the name and address of an employee. The application uses Excel as a client to this server to retrieve the list of employees and display their details. The user then enters the salary for each employee; Excel calculates the tax payable on that salary. When the user closes the spreadsheet, Excel invokes the `setAnnualSalary` and `setTaxPayable` operations on each employee to update the server.

Figure 5 shows the spreadsheet after start-up. The highlighted cells are filled with the details that are returned by the Ice server.

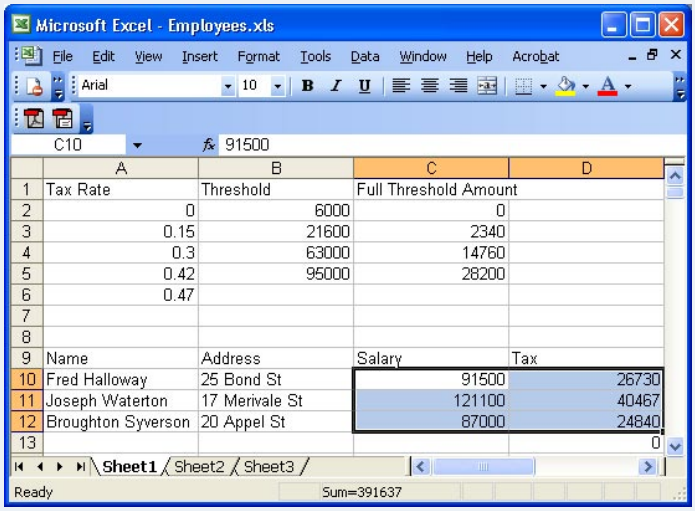
Figure 5: Initialized Spreadsheet



The user enters the salary for each employee in column C, and Excel calculates tax payable for each employee in column D. The result is shown in Figure 6; the highlighted cells show the salary entered by the user and the tax calculated by Excel. When the user

closes the spreadsheet, Excel sends the content of these cells to the Ice server.

Figure 6: Initialized Spreadsheet



	A	B	C	D
1	Tax Rate	Threshold	Full Threshold Amount	
2	0	6000	0	
3	0.15	21600	2340	
4	0.3	63000	14760	
5	0.42	95000	28200	
6	0.47			
7				
8				
9	Name	Address	Salary	Tax
10	Fred Halloway	25 Bond St	91500	26730
11	Joseph Waterton	17 Merivale St	121100	40467
12	Broughton Syverson	20 Appel St	87000	24840
13				0

Now on to the code… (I am only discussing the client code here because the server code is not relevant to this topic. But, of course, you can download the complete code for this article from our web site.) A Visual Studio Excel project produces a DLL that is called by Excel, provided that the spreadsheet and your machine are configured correctly. (I will discuss the configuration issues shortly.) The DLL that is called by Excel contains a class with two methods, `ThisWorkbook_Open` and `ThisWorkbook_BeforeClose`. Your application code forms the body of these two methods. Visual Studio creates an outline of this code for you when you create an Excel project:

```
Imports Excel = Microsoft.Office.Interop.Excel
Imports System
Imports Employees

' Office integration attribute.
' Identifies the startup class for the workbook.
<Assembly: System.ComponentModel. _
    DescriptionAttribute("OfficeStartupClass, _
    Version=1.0, Class=OfficeCodeBehind")>

Public Class OfficeCodeBehind

    Friend WithEvents ThisWorkbook _
        As Excel.Workbook
    Friend WithEvents ThisApplication _
        As Excel.Application

    ' Generated code here...

    ' Called when the workbook is opened.
    Private Sub ThisWorkbook_Open() _
        Handles ThisWorkbook.Open
        ' Your start-up code here...
    End Sub

    ' Called before the workbook is closed.
    ' Note that this method
    ' might be called multiple times and the
    ' value assigned to Cancel might be ignored
    ' if other code or the user intervenes.
    ' Cancel is False when the event occurs.
    ' If the event procedure sets this to True,
    ' the document does not close when the
    ' procedure is finished.
    Private Sub ThisWorkbook_BeforeClose( _
            ByRef Cancel As Boolean) _
            Handles ThisWorkbook.BeforeClose
        ' Your finalization code here...
    End Sub
End Class
```

To turn our spreadsheet into an Ice client, we need to flesh out this class. To start with, we add a few private data members to the class, which we initialize in `ThisWorkbook_Open`. These data members store the active worksheet, the Ice communicator, the sequence of Employee proxies returned by the `list` operation, and the row index at which we will fill the spreadsheet with employee details.

The `ThisWorkbook_Open` method initializes the Ice run time, creates a proxy to the `Finder` object, and calls `list` on that object to obtain the sequence of `Employee` proxies, which it stores in the `employees` data member. The code then enters a loop in which it retrieves the details for each employee and fills the corresponding cells in the spreadsheet with these details.

```
Private sheet As Excel.Worksheet
Private communicator As Ice.Communicator
Private employees As EmployeeSeq
Private Const firstRow = 10 ' Employees start
                            ' on row 10

Private Sub ThisWorkbook_Open() _
        Handles ThisWorkbook.Open
    Try
        ThisApplication.Visible = True
        ThisApplication.UserControl = True
        sheet = ThisWorkbook.ActiveSheet

        Dim args As String() = {""}
        communicator = Ice.Util.initialize(args)

        Dim prx As Ice.ObjectPrx = _
                communicator.stringToProxy( _
                    "Finder:tcp -p 10000")
        Dim finder As EmployeeFinderPrx = _
                EmployeeFinderPrxHelper. _
                    uncheckedCast(prx)

        Dim row = firstRow
        employees = finder.list()
        For Each emp As EmployeePrx In employees
            Dim details As EmployeeDetails = _
```

```
                emp.getDetails()
            sheet.Cells(row, 1) = details.name
            sheet.Cells(row, 2) = details.address
            row += 1
        Next
    Catch ex As Exception
        If Not communicator Is Nothing Then
            Try
                communicator.destroy()
            Catch
            End Try
            communicator = Nothing
        End If
    End Try
End Sub
```

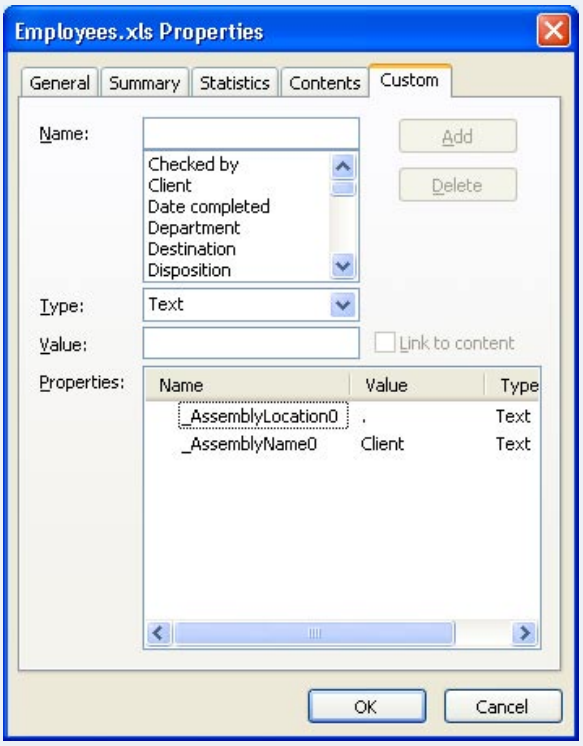
Once the user closes the spreadsheet, Excel calls the `ThisWorkbook_BeforeClose` method. Note that the method sets `Cancel` to `False` before returning. This ensures that Excel will actually close the spreadsheet; if you set `Cancel` to `True`, the spreadsheet stays open. You can use this functionality to perform data validation and present a dialog with an error message if anything is wrong. For this simple application, we do not perform any validation, but simply extract the salary and tax details from the spreadsheet and update them in the Ice server before shutting down.

```
Private Sub ThisWorkbook_BeforeClose( _
        ByRef Cancel As Boolean) _
        Handles ThisWorkbook.BeforeClose

    If Not communicator Is Nothing Then
        Try
            Dim row As Integer = firstRow
            For Each emp As EmployeePrx _
                    In employees
                emp.setAnnualSalary( _
                    Double.Parse( _
                       sheet.Cells(row, 3).Value))
                emp.setTaxPayable( _
                    Double.Parse( _
                       sheet.Cells(row, 4).Value))
                row += 1
            Next
            communicator.destroy()
        Catch
        End Try
        communicator = Nothing
        Cancel = False
    End If
End Sub
```

This is all there is to writing the code for our Excel client. However, before you can launch the spreadsheet and have it talk to the Ice server, you must take care of a few configuration issues. As it turns out, this is somewhat complex (at least when you do it for the first time), so I'll discuss the configuration step by step. I also recommend that you read Brian Randell's and Ken Getz's article on deploying Visual Studio Tools for Office applications, which provides a good overview of how Office applications determine whether to trust a particular piece of code.

The first step is to ensure that your spreadsheet actually knows that it is supposed to execute your code when it is launched. To do this, you must set two properties in the spreadsheet, `_AssemblyLocation0`, which determines in which directory your code can be found, and `_AssemblyName0`, which determines the name of the DLL that Excel should call. In this case, the name of the DLL is `Client.dll`. Figure 7 shows the Properties dialog of the spreadsheet in which you can set these properties. In this example, the spreadsheet looks for the assembly containing your code in its current directory. (For a real-world application, you would use a full path name and place the assembly in a directory that cannot be written to by ordinary users.)

Figure 7: Setting Excel Properties



When Excel starts up, it loads the assembly that is specified by the `_AssemblyName0` property. The assembly, in turn, contains an attribute `OfficeStartUpClass`, which provides the name of the class (`OfficeCodeBehind`, in this case) that contains your `ThisWorkbook_Open` and `ThisWorkbook_BeforeClose` methods. (You can see the corresponding attribute being set at the start of the excel client code on page 12.) So, Excel knows which assembly contains your code by looking at its `_AssemblyName0` and `_AssemblyLocation0` properties, and it knows which class contains your code by looking at the `OfficeStartUpClass` attribute inside the assembly. However, Excel will not execute your

code unless the .NET Framework determines that Excel can actually trust that code. (If your code is not trusted, you get an error message during Excel start-up telling you so.)

To get .NET to conclude that your assembly should be trusted for the purpose of being executed inside an Office application, you must add code groups to the .NET configuration. Each code group determines which assemblies belong to the group, and what level of trust should be granted to the assemblies in the group. In other words, code groups are a mapping of assemblies to sets of permissions.

The .NET Framework uses a number of different permission sets, each of which represents a particular level of trust. (There are permission sets that grant no permissions at all, grant execute permission only, and so on.) For Excel to execute the code in your assembly, your assembly must have FullTrust (which is the name of one of the permissions sets). In addition, any assemblies that are loaded by your assembly (such as the Ice run time) also must have full trust.

So, to complete our configuration, we must tell the .NET Framework that your assembly and the Ice run time are to be trusted when executed by an Office application. There are several ways to configure the .NET framework: using a GUI tool, via the command line (using `caspol.exe`), and programmatically (via a number of classes in the `System.Security` namespace). For this example, I will show how to do this using the GUI tool. You can launch the tool from the Control Panel, by selecting "Administrative Tools" and "Microsoft .NET Framework 1.1 Configuration."

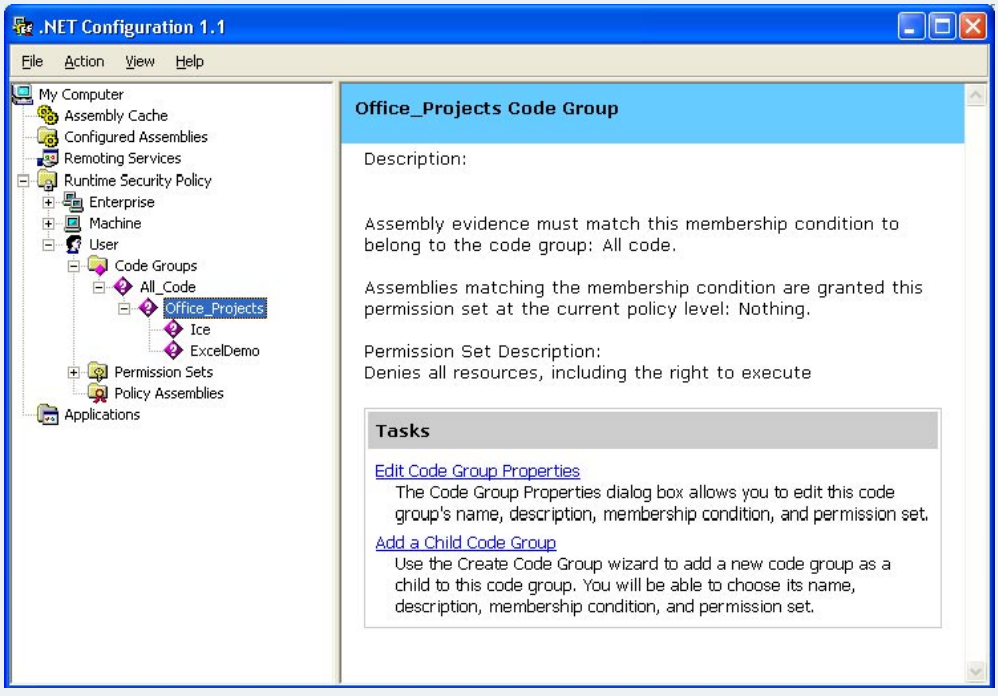
Figure 8 shows a screen shot of this program.

.NET determines trust at a number of levels: the user, local machine, enterprise, and application domain level. In addition, trust can be established based on the zone from which an assembly originates. For example, you can assign different trust to assemblies on the local machine and assemblies on the local intranet. Excel not only requires full trust, but also requires full trust at the application domain level so, for this example, we configure .NET at the application domain level.

As you can see in Figure 8, within the user level, we have a node called "Code Groups", with a sub-group called "All Code". If you examine the settings for the "All Code" group, you will find that it has full trust. However, because the Office loader applies trust at the application domain level, this is not sufficient. Underneath the "All Code" node, the Office loader looks for a node called "Office_Projects"; child nodes underneath this node establish the required application domain-level trust. To get Excel to trust our assembly, we need two nodes here, one that grants trust to the client assembly, and one that grants trust to the Ice run time (because the client assembly loads the Ice run time).

To make matters more interesting, you can establish trust based on a number of membership conditions. For example, you can state that an assembly is to be trusted based on its location in the file system, its hash code, or its strong name (among other choices). For this example, we will establish trust for the client assembly based on its location, and establish trust for the Ice run time based on its strong name.

Figure 8: .NET Configuration Tool



To configure trust for the client assembly, we add a code group underneath the "Office_Projects" group. You have to assign a name to the code group ("ExcelDemo", in this case), specify a membership condition, and specify a permission set (which must be "FullTrust"). Figure 9 shows the dialog that specifies the membership condition.

The condition type is set to "URL". By specifying a file: URL, you can determine for which assembly in the file system trust is to be granted. Note that the path name ends in a wildcard, meaning that all assemblies in the specified directory will be trusted. (You can also target a specific DLL by providing its full path name.)

Figure 10 shows the dialog that specifies trust for the Ice run time. In this case, trust is based on the assembly's strong name, that is,

Figure 9: Membership Condition for Client Assembly

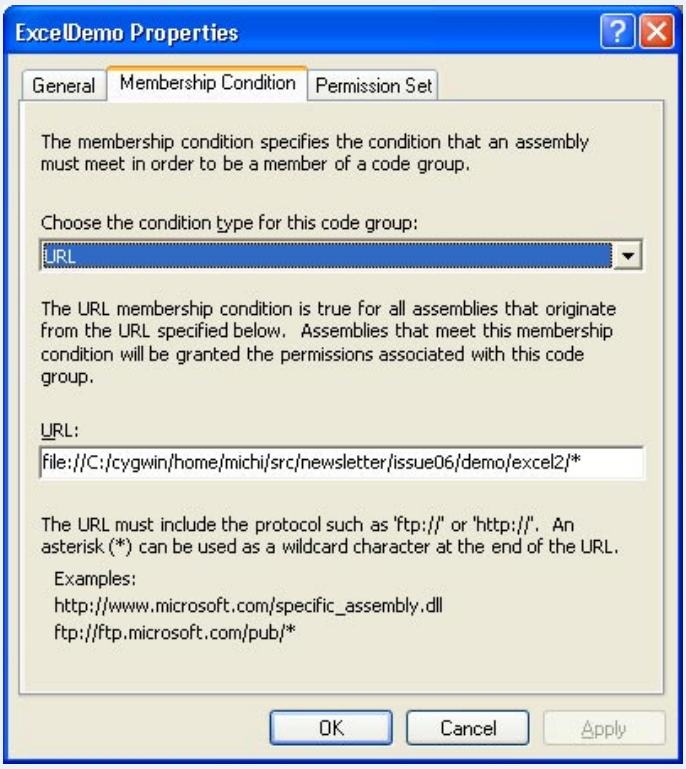
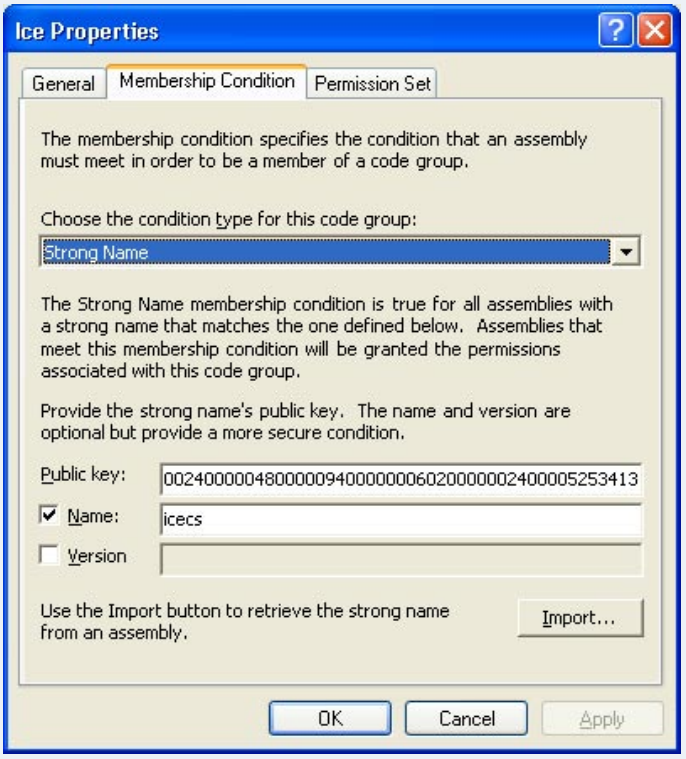


Figure 10: Membership Condition for the Ice Runtime



its public key. In addition, you can also specify the name of the assembly and, if desired, a specific version of the assembly. Note that, with this form of trust, it does not matter where the assembly is located—the usual algorithm to locate assemblies is used by .NET and, once the assembly is found, the run time checks whether it is to be trusted or not, based on the membership condition and permission set. Note that using the strong name of an assembly to establish trust is particularly useful if the assembly is installed in the Global Assembly Cache (GAC).

Summary

The technique I outlined in this article does not only apply to Excel, but also to Word, so you can exchange data between Word and an Ice server in the same fashion. Application integration in this way is not only easy to achieve, it can also be immensely useful for a variety of business processes, such as updating sales figures, checking stock levels, reporting field data back to head office, or, as for this article, computing tax.

Excel also provides a number of other features that you can use in your program, such as interactive dialogs, meaning that using Excel as a client front end can save you a lot work: instead of writing a complete application from scratch, you can reuse a lot of the features that are already built into Excel. Keep this idea in mind for the next time you are asked to write a simple client to access remote data: by reusing existing applications as much as possible, you can avoid reinventing the wheel in these taxing times...

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: Does Ice catch any signal?

This answer applies to Ice for C++ on all platforms except Windows.

During the creation of the first communicator in a process, Ice changes the signal disposition of `SIGPIPE` to `SIG_IGN` (ignore). When the last communicator in a process is destroyed, Ice changes `SIGPIPE`'s disposition to `SIG_DFL`; the default action for `SIGPIPE` terminates the process without core dump.

Ice does not interact with any other signal; in particular, Ice does not install any signal handler. That said, Ice also provides helper classes with signal-handling capabilities:

- `IceUtil::CtrlCHandler` handles `CTRL-C` and `CTRL-C`-like signals (`SIGINT`, `SIGTERM` and `SIGHUP`) by calling a function registered by the application. See the "Portable Signal Handling" section in the Ice manual for details on this class.
- `Ice::Application` and `Ice::Service` each use a `CtrlCHandler` to shut down or destroy the communicator when the process receives a `CTRL-C` or similar signal.

These three classes are helper classes: they simplify coding, but you do not have to use them. In particular, if `Ice::Application` or `Ice::Service` do not handle `SIGINT`, `SIGTERM` and `SIGHUP` in the way you like, you can implement your Ice application without their help.

Q: Why do I get an Ice::UnknownException?

`UnknownException` has two subclasses `UnknownUserException`, and `UnknownLocalException`.

- `UnknownUserException` is raised if the server implementation throws an Ice user exception that isn't in the exception specification for the operation.
- `UnknownLocalException` is raised if any Ice run-time exception (any exception derived from `LocalException`) other than `ObjectNotExistException`, `FacetNotExistException` and `OperationNotExistException` is thrown by the server implementation.
- `UnknownException` is raised if any other exception is thrown by the server implementation. Typically, the server will give more information in the debug log if the property `Ice.Warn.Dispatch` has a non-zero value. Some language mappings, such as Java, also provide comprehensive information in the `reason` string.

If you have a C++ server under Windows that raises an `UnknownException`, this often means the server has suffered an access violation, causing the C++ run time to raise a structured exception that is later caught by a "`catch(...)`" handler in the Ice dispatch code. During debugging with Visual C++, you can trap such exceptions by setting "Debug/Exceptions/Win32 Exceptions/Access violation" to "Stop always" instead of "Stop if not handled."

You might wonder why there is an `UnknownLocalException`. The Ice run time knows all of the possible run-time exceptions, so why does it not return the true error to the caller? The primary reason is security: the Ice run time will not tell the client more than it needs to know. Another reason is that the client cannot do anything meaningful with more information. From a client-side perspective, you are interested in only two outcomes: did it work, or did it not work? If it didn't work, you need to know whether the problem is a client-side or a server-side issue. The run-time exceptions that are returned to the client allow you to make this decision; other details about the exception are a server-side concern, and therefore not provided to the client.

Q: What are the various forms of stringified proxies?

There are two types of stringified proxies, direct proxies and indirect proxies.

Direct proxies have the form <*identity*> `:`<*endpoints*>. In turn, the stringified form of an object identity is `<category>/<id>`. The category is optional and, if not present, the category is the empty string. The endpoints comprise a colon-separated list of possible addresses for the server. Each address has a protocol identifier, followed by protocol-specific addressing information. Currently, Ice supports `tcp`, `udp`, and `ssl`. The addressing information for each of these protocols is the same: `-h <host>` and `-p <port>`. When a client makes the first invocation on a proxy, the Ice run time tries to bind to each of the endpoints in a random order. If binding to all of the endpoints fails, the run time raises an exception (in most cases, a `ConnectFailedException`.)

Indirect proxies have the form <*identity*>`@`<*adapter-name*>, with the adapter name being optional. Note that an indirect proxy contains no addressing information: you cannot specify both an adapter name and a list of endpoints, that is, a proxy of form <*identity*>`@`<*adapter*>`:`<*endpoints*> is syntactically invalid. To get a list of addresses from an indirect proxy so it can bind to the target object, the client-side run time consults a locator, the address of which is controlled by the `Ice.Default.Locator` property, and asks the locator for the addresses at which an object with the given identity and adapter name can be found.