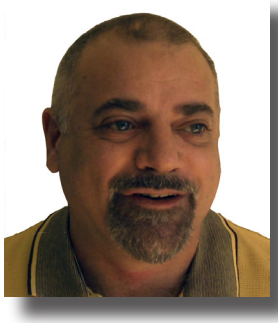
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 22, April 2007



Good Programmers are Rare

This issue represents something of a milestone for *Connections*: for the first time, we are publishing an article that was not written by one of ZeroC's staff, but by a customer. In this issue, you will find An Autonomous Vehicle Using Ice and Orca, by Alexei Makarenko, Alex Brooks, and Ben Upcroft, who are members of the Orca Project. In November, the Orca project will send a real car into a competition that requires the car, by itself, to drive for sixty miles through a cityscape complete with traffic signs, lane markings, obstacles, and other moving traffic. Now if that isn't cool, I don't know what is... And the mind boggles at the complexity of the task: the car has to "see", plan a route, and follow the route without bumping into other cars, all while obeying traffic rules—Orca obviously has some first-rate programmers, and we wish them good luck!

On the subject of programmers, we all know that there are good and not so good programmers: it is not unusual to hear reports of programmers who are ten times more productive than their colleagues on the same team. These are the programmers who are akin to miracle workers—the much-fabled "super-hackers". So, what is it that these programmers have that their colleagues don't?

Most programmers would agree that much of their work is fundamentally a creative activity. Coming up with good solutions to tricky problems requires creativity, lateral thinking, being adventurous, and finding that new angle on a problem that other people cannot see. In other words, being a good programmer requires flair and, at the top level, is more of an art than a craft. (I believe it is no coincidence that Donald Knuth called his famous series of books *The **Art** of Computer Programming*.) In terms of the Myers-Briggs Type Indicator, people with such artistic and lateral-thinking qualities are known as STPs. About 10% of all people fall into this category—not that many.

Another quality that good programmers require is fanatic attention to detail. Programs are supremely unforgiving, and the most innocent omission or inaccuracy can have catastrophic consequences. For example, in 1999, the Mars Climate Orbiter was lost because a program produced thrust data in imperial units; that data was passed to a trajectory calculation program that expected metric units. Similarly, in 1996, an Ariane 5 was lost because the code converted a 64-bit floating-point number into a 16-bit integer, causing overflow. (The story about Mariner 1 being lost due to a period instead of a comma in a FORTRAN DO-loop is fun to tell, but not true.) These incidents show rather dramatically that programs have zero tolerance for "Oh, come on, you *know* what I meant!" So, good programmers are perfectionists. They cross all the t's and dot all the i's, and they have the stamina to keep going long after mere mortals have decided things are good enough. In terms of Myers-Briggs again, people with such meticulous attention to detail are known as NTJs. About 4% of all people fall into this category—not many.

What are the chances of finding a good programmer? There are not that many STPs, and there are even fewer NTJs. But the real problem is that the two qualities that are required for good programming are diametrically opposed to each other. People who are artistic and creative lateral thinkers are usually not terribly good at putting the lid back on the tube of toothpaste after brushing their teeth. And, similarly, people who have the ability to check, test, and re-check everything several times before they are satisfied are usually not inclined to paint beyond the edge of the canvas. In other words, the intersection of the sets of STPs and NTJs is very close to empty. Or, to put it bluntly, how many creative, lateral-thinking, anal-retentive perfectionists do you know? I don't know many—good programmers are a rare breed indeed!

Michi Henning
Chief Scientist

Issue Features

An Autonomous Vehicle Using Ice and Orca

The Orca team discuss how they use Ice in the urban challenge competition.

Teach Yourself Glacier2 in 10 Minutes

In this article, Michi Henning gives a concise overview of how to use Glacier2.

Contents

An Autonomous Vehicle Using Ice and Orca

Alexei Makarenko, BSD Team Software Architect, Orca Project Administrator
Alex Brooks, Orca Project Administrator
Ben Upcroft, BSD Team Co-Leader

Australian Centre for Field Robotics

Introduction

The Urban Challenge sponsored by the US Defense Advanced Research Projects Agency (DARPA) is today's largest competition in the field of mobile robotics. This year's challenge is to build an autonomous vehicle capable of traversing 60 miles of mock urban area as fast as possible while obeying speed limits and other traffic rules. Aside from emergency shutdown, no human intervention is allowed once the car is on its way. The competition will be held on the west coast of the US in November 2007.

This article is about the software used in the robotic car being built by one of over sixty registered teams. The Berkeley-Sydney Driving Team brings together researchers and students from Australia and the United States. The Australian side is headed by the Australian Centre for Field Robotics from the University of Sydney and the American side is represented by University of California, Berkeley.

Figure 1 shows our team's car, a modified Toyota RAV4 sports utility vehicle. Custom actuators were installed to allow computer control of accelerator position, braking, and steering. Environmental sensors such as cameras and lasers are placed on the roof. The computer rack is located in the back of the car.

The goal of the competition is to navigate through a series of checkpoints specified with GPS coordinates. This has to be done while driving safely and obeying traffic rules such as staying in lanes, not exceeding speed limits, safely passing other cars, crossing intersections, etc.

Algorithmically, this task translates into the ability to make and execute plans that bring the car from point A (the current location) to point B. In order to make that happen, we must extract information from the on-board sensors about the environment and the vehicle itself. Some of the representative sub-tasks include tracking the vehicle location relative to the map; identifying street boundaries, buildings and other cars; controlling vehicle motion; reasoning about traffic rules, and many others.

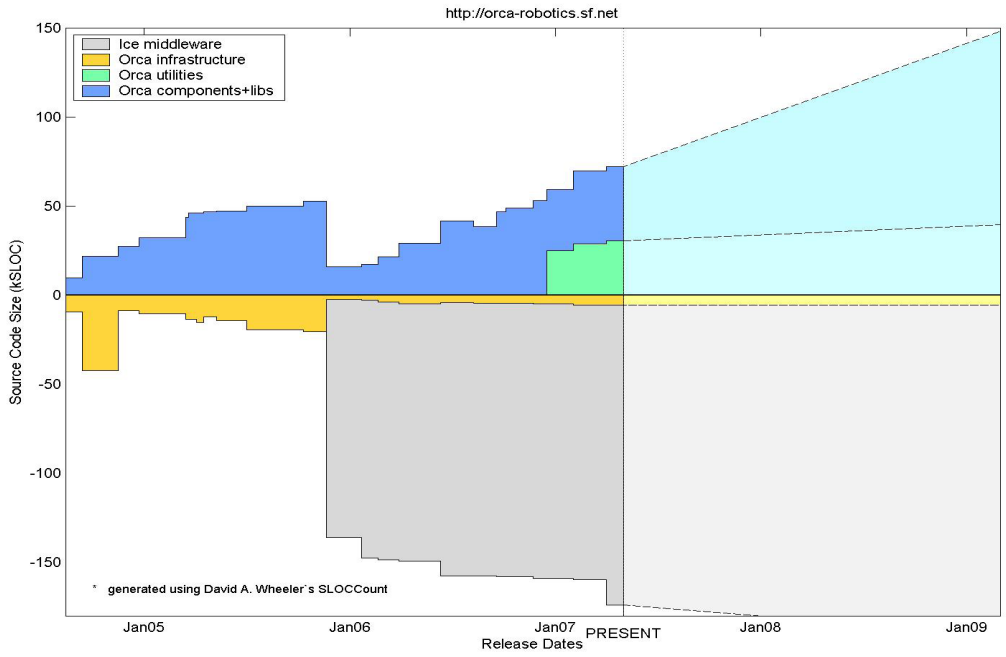
The Software Architecture

Building reliable robotic hardware is a difficult task in itself, but the major challenge of this competition is in software. The main difficulties arise from the task's complexity and its dynamic real-time nature. Other contributing factors include the distributed and cross-platform computing environment, the large number of software contributors, and the need to use existing code.

Our software is built using a Component-Based Software Engineering (CBSE) approach. This offers modularity, software reuse, and flexibility in deployment, all of which are necessary to address the problems listed above. Applied to a robotic application,

Figure 1: The customized Toyota RAV4 during initial field trials.

Figure 2: Source code size of the Orca project. Projection into the future illustrates our goal of concentrating on writing robotic software and relying entirely on Ice for all middleware functions.



Source code size gives a sense of a project's activity and trends. Figure 2 shows the history of line-count statistics for the Orca project, generated using David A. Wheeler's SLOCCount. This figure helps to illustrate Orca's main objective: we are interested in a large "superstructure" (blue, representing useful components) and do not want to "dig a deep foundation" (orange, the infrastructure necessary to allow components to talk to each other). This was the main motivation for switching to Ice at the end of 2005. Doing so not only reduced the size of the Orca infrastructure, but it also stopped a troublesome trend of infrastructure growth driven by the need for features beyond basic communication. Projecting this plot into the future, we hope to see the current trend continue: keeping our infrastructure code very thin, while increasing the repository of reusable components.

CBSE means that the algorithms mentioned in the introduction are mapped to a set of components. Components run asynchronously and exchange information through communication.

We use Ice middleware extensively in our system: for component interface definition, inter-component communication, component deployment, location, activation services, etc. We also use Orca—an open source project that customizes Ice to robotic applications and provides an on-line repository of reusable components. In the remainder of the article we focus on two areas:

- what Orca is and what it adds to Ice, and
- how we use the Ice/Orca combination in the control of an autonomous vehicle.

The Orca Project

Lack of reliable reusable software for robotics is a well-recognized problem as evidenced by several active standardization efforts in robotic software, particularly in academia. The Orca project traces its roots to one of them: the EU-funded OROCOS project, which was started in 2001 to develop open robot control software. At that time, the component model was implemented with CORBA using ACE/TAO. Since then, the source code has been re-written several times, the project has moved to SourceFORGE, the project's name has changed, and the center of the development effort has moved from Europe to Australia. Work on porting Orca to Ice 2.1.2 began in March 2005.

Given the fact that Orca is based on Ice, what does Orca actually add to the infrastructure? There are three main areas of contribution: common Slice definitions; an optional convenience library in C++; and a repository of reusable components, libraries, and utilities. Here we list and illustrate some of the useful things found in the Orca distribution.

1. **Slice definitions for data types frequently encountered in robotics such as time, coordinate frames, kinematic and dynamic elements, etc.**

```
// Slice
module orca
{
    // Unix time
    struct Time
    {
        int seconds;
        int useconds;
    };

    // 2D position in Cartesian coordinate
    // system [m]
```

```
    struct CartesianPoint2d
    {
        double x;
        double y;
    };

    // 2D Cartesian coordinate frame: origin
    // and orientation [m,rad]
    struct Frame2d
    {
        CartesianPoint2d p;
        double o;
    };
};
```

These definitions are extremely simple, that is, anyone can write them with minimum effort. The real value of such definitions is that they provide common representations and make interoperability between components possible.

2. **Slice definitions of interfaces that are frequently encountered in robotics.** As an example, we show `Odometry2d`—an interface to the vehicle motion data in two dimensions, typically collected by measuring wheel rotation. Here we use the Slice structures defined above. (Exception definitions are not shown.)

```
// Slice
module orca
{
    struct Odometry2dData
    {
        Time timeStamp;
        Frame2d pose;
        // ...
    };

    interface Odometry2d
    {
        idempotent Odometry2dData getData()
            throws DataNotExistException,
                HardwareFailedException;
        // ...
    };
};
```

Slice definitions of interfaces are a key to interoperability, both inside and outside the Orca framework. For example, it would be quite possible to write software without using Orca (by using Ice directly, for example) that would still be fully compatible with existing and future Orca components, as long the same interface definitions are used.

3. **An optional C++ library that contains convenience classes and macro-like functions to simplify common usage.** For example, there are two commonly used component container classes: `orcaice::Application` (derives from `Ice::Application`) and `orcaice::Service` (derives from `IceBox::Service`). Both contain a pointer to `orcaice::Component`. If an Orca developer implements a new component by deriving from `orcaice::Component`, then, with the help of `libOrcaIce`, the component can be easily deployed either as a stand-alone application or as an IceBox service.

4. **Optional tools and utilities, such as a data visualization GUI, a data log/replay facility, and so on.** These tools can be thought of as domain-specific services that operate on the level of Orca interfaces. Because of this, new components using standard interfaces can take advantage of the tools that have already been written. For example, a new component that represents an eight-wheel vehicle and provides the standard `Odometry2d` interface can display and log its data without any additional effort. The benefit of this type of software reuse is reflected in Figure 2—the expected rate of growth in the Orca utility code is lower than the rate of growth of the components.

The Urban Challenge Software

Let us return to the software implementation of the autonomous vehicle. The total number of components that will comprise our final system is still unknown. (For reference, the winner of the previous competition had about thirty.) The on-board computer system currently has four hosts (not counting diagnostic laptops that are often connected to the system), and this number is likely to grow. The on-board computers use two operating systems: Ubuntu Linux and QNX. The software is written by about a dozen people from four organizations (this number is higher if we count the authors of the existing components used directly in our system).

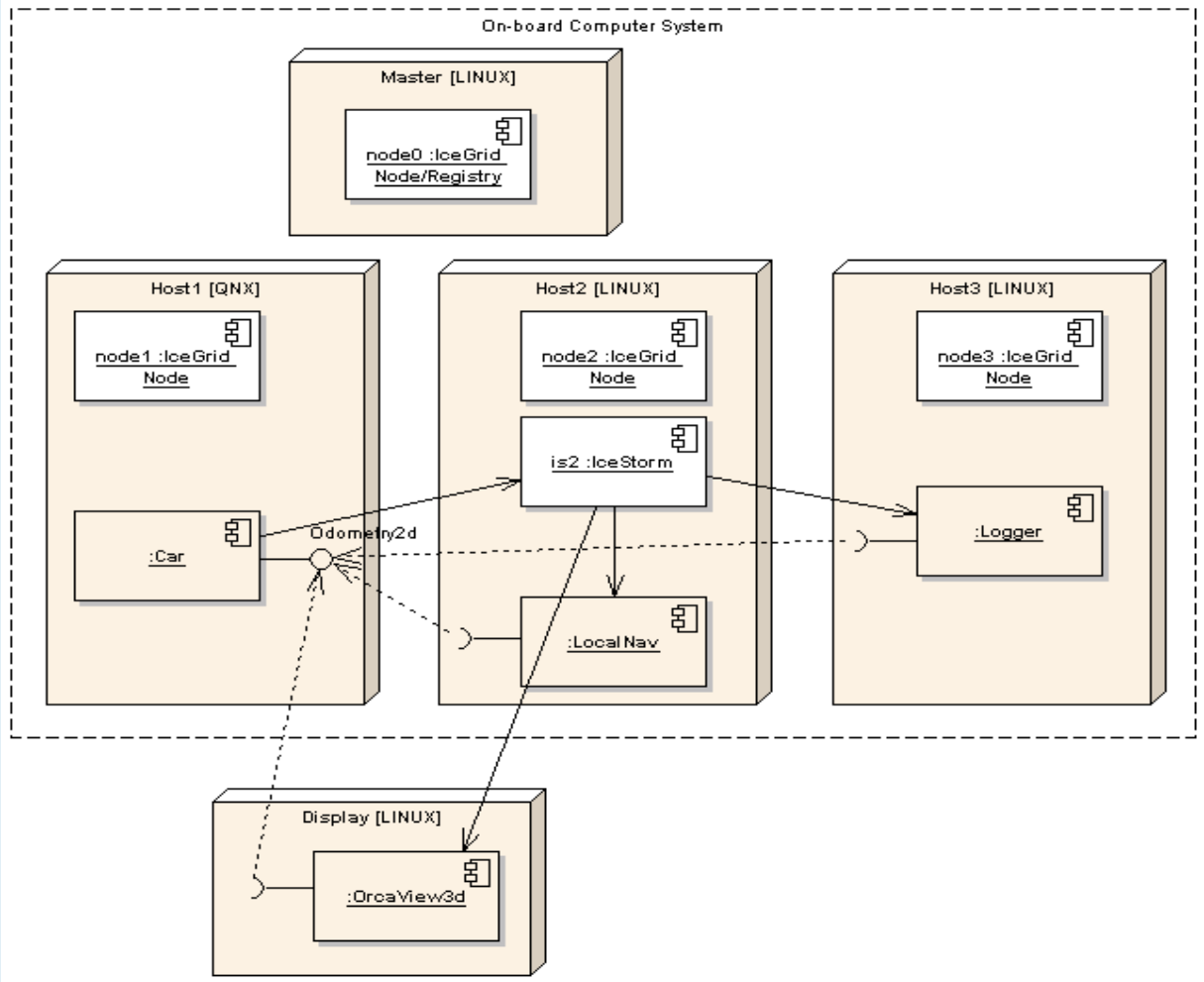
The computing hardware uses off-the-shelf rack-mounted PCs with Intel dual-core processors. The hosts are connected with a standard 1-Gigabit Ethernet hub. We have done some performance tests with this setup; the Orca web site provides these latency figures.

Some but not all parts of our system require real-time features. For example, there is a strong need for accurate time-stamping of sensor data. A vehicle in the competition moves at speeds of up to 30mph (48km/h) and small sporadic delays in the standard Linux kernel can have significant negative impact, particularly in navigation. We use a dedicated host running the real-time QNX Neutrino operating system for all interactions with sensor and actuator hardware. Our own (partial, unsupported) port of Ice to QNX is available through the ZeroC developer forums.

Figure 3 illustrates the current deployment strategy. The master host executes an instance of an IceGrid node with a collocated IceGrid registry. Every other host runs an IceGrid node. One host runs QNX Neutrino, the remaining hosts run Ubuntu Linux. The QNX host executes low-level actuator and sensor components. Deployment of other components has not been finalized; however, the flexible nature of a component-based architecture does not require us to make difficult deployment decisions *a priori*.

Some components that we use in the vehicle are currently publicly available, such as the laser range-finder, the inertial

Figure 3. Deployment diagram illustrating how the on-board computer system is configured with IceGrid, include the use of IceStorm for data distribution.



navigation system, and others; due to the competitive nature of this project, other components, such as the `Car` component in Figure 3, are not publicly available. However, we intend to release most of these components through the Orca project in the future.

Our system uses several Ice services. Here we give brief notes on our experiences and the lessons we have learned.

Software Deployment

For software deployment, we rely heavily on IceGrid. The ability to manage the entire system from a single XML file greatly simplifies configuration management. The IceGrid GUI is invaluable in providing feedback about the state of the system. We find that, when testing the car, the GUI is on the screen of the test engineer's laptop the entire time. For an added measure of reliability, we are also planning to use the registry fall-back functionality that became available in Ice version 3.2.

Software distribution

Software updates are frequent during the development of a system such as ours, which makes software distribution a frequent (and tedious) task. We use the following process: the software source is updated from CVS and compiled on the master host (see Figure 3); then the new binaries are pushed out to all Linux hosts using IcePatch2. QNX binaries are compiled separately.

Data distribution

Many low-level robotic components fall into the category of device drivers that interact with hardware such as lasers, wheel encoders, GPS receivers, and so on. The data generated by these devices typically needs to be distributed to several clients—exactly the function that IceStorm is designed to provide. One design decision we had to make is how to direct clients to the correct IceStorm server and topic. We could have done this with additional configuration parameters on the client side or by using similar conventions. However, we found that the following pattern is a cleaner option: the device interface (the server side) performs the subscription for the clients. To illustrate, let us continue the example of the `Odometry2d` interface:

```
// Slice
module orca
{
    interface Odometry2dConsumer
    {
        void setData(Odometry2dData obj);
    };
```

```
    interface Odometry2d
    {
        // ...
        void subscribe(
            Odometry2dConsumer* subscriber)
            throws SubscriptionFailedException;
        idempotent void unsubscribe(
            Odometry2dConsumer* subscriber);
    };
};
```

In our implementation, the `Odometry2d::subscribe` function simply contacts the appropriate IceStorm server and subscribes the client to the appropriate topic. The location of the IceStorm server, the particular naming convention for the topic, and the quality of service settings are all chosen by the server. All of this information (and even the fact that we are actually using IceStorm) is transparent to the client. The pattern is illustrated in Figure 3. Several clients connect to the `Odometry2d` interface of the `Car` component and subscribe themselves to the data stream. (They have to implement the `Odometry2dConsumer` interface, of course.) The `Car` component subscribes the clients to a topic whose name and server is determined by the component. The published data then flows from the `Car` component through the IceStorm server to the clients.

Summary

Among many challenges presented by the DARPA competition, software complexity is one of the most difficult. (This statement may also be true for the field of mobile robotics as a whole.) The component-based approach helps manage this complexity by breaking up a monolithic implementation into manageable parts.

While the advantages of modularization are widely recognized, it is also true that the details of inter-component communication are far from trivial and result in extra complexity that can easily outweigh the benefits of the modular solution. In this context, we find that Ice middleware is a great enabling technology that unburdens the component developer from the nitty-gritty details of communication.

The DARPA Urban Challenge is a good motivator for the field of robotics in general, and for the Orca community in particular. Through the experience we gained in this project, we have already improved the Orca framework; beyond the competition, we hope that the Orca project will lead to more cooperation among academic institutions. The project may even provide a bridge to the robust commercial solutions that we feel are necessary for continuing progress in this field.

Teach Yourself Glacier2 in 10 Minutes

Michi Henning, Chief Scientist

Introduction

Issue 19 of *Connections* contained *Teach Yourself IceGrid in 10 Minutes*, which provides a gentle introduction to getting started with IceGrid. We received a fair amount of positive feedback on that article so, seeing that I'm onto a good thing, I decided to continue in the same vein for other features of Ice. Of course, all this is despite me, in the same article, having slagged off books that claim to be able to teach something worthwhile in ten minutes. I stand by my opinion: if you want to learn anything non-trivial about a computing topic, you will have to invest more than ten minutes, and Glacier2 is no exception, despite the title of this article. But getting acquainted with Glacier2 really does take only a few minutes. (Well, yes, OK, a little more than ten minutes, maybe twenty or thirty…)

So, I will continue to live in a state of schizophrenia and write articles entitled *Teach Yourself <Something> in 10 Minutes*, while preaching that people cannot learn anything worthwhile in that time—go figure… (And, yes, I will get around to writing that editorial eventually!)

What is Glacier2?

Glacier2, in a nutshell, is a firewall traversal service for Ice: it allows Ice servers to sit behind a corporate firewall, such that clients in the outside world can use these servers. Glacier2 is a simple program; at its core, Glacier2 is an Ice server that receives incoming requests from clients and passes them on as blobs of bits to servers. This is quite similar to the functioning of an IP router that receives packets on one interface and forwards them via another interface. This simplicity not only makes Glacier2 easy to configure, but it also makes it much more likely that Glacier2 is secure. (Lower complexity means fewer bugs, not to mention better performance.) In particular, Glacier2 does not depend for its security on the integrity and correct configuration of other components, such as a web server. (Web services, anyone?) Glacier2 can also enforce that clients can connect to servers only via SSL, but not via insecure TCP connections. (Glacier2 does not support UDP.)

Feature Highlights

Here are some of the main distinguishing features of Glacier2:

- You only need to open a single port in the corporate firewall for any number of Ice servers behind the firewall.
- Glacier2 can be configured to only accept SSL connections.
- Glacier2 does not require any configuration that would need to change as applications change. In particular, Glacier2 does *not* require knowledge of the Slice definitions used by the back-end servers.
- Clients require only minimal source code changes in order to work with Glacier2.
- Servers do not require any source code changes in order to work with Glacier2.
- Callbacks from servers to clients do *not* require the client's firewall to permit incoming connections.

In addition to the above highlights, Glacier2 also offers a number of advanced features:

- Access control, which allows you to add additional security controls beyond those provided by SSL, such as authentication with passwords or SSL certificates.
- Integration hooks for custom authentication mechanisms.
- Filters that allow you to restrict which addresses and ports a client can access on the internal network. Filters can also be used to limit client requests to specific object adapters or objects.
- Session management, which allows Glacier2 to recover resources associated with clients that do not disconnect in an orderly fashion. This includes hooks that you can use to integrate Glacier2's session management with application-specific functionality, for example, to establish and clean up per-client contextual information.

As befits an introductory article, I will focus on getting started with Glacier2 and will leave you to check the Ice Manual for details on the advanced features.

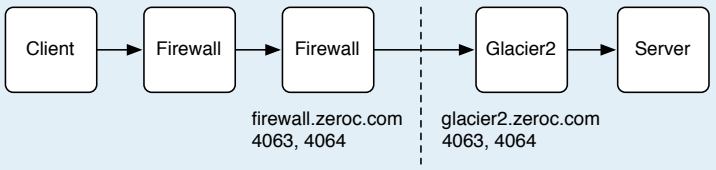
Getting Started

Configuring Your Firewall

Chances are that you will already have a firewall that is configured to disallow incoming connections (except for a number of selected services, such as web and email traffic). To allow Glacier2 to work with your firewall, you must configure the firewall to open a single TCP port and forward all traffic for that port to the machine on which Glacier2 runs. Exactly how you achieve this depends on your firewall. However, most firewalls have an administrative interface that allows you to easily add a rule that essentially says "forward all incoming TCP traffic on port 4064 to port 4064 on machine glacier2.zeroc.com." Figure 1 illustrates this situation.

We suggest that you use port 4064 as the incoming SSL port for Glacier2 and, if you want to allow client access via TCP, that you use port 4063 as the incoming TCP port for Glacier2. These two ports are reserved for Glacier2 by IANA (Internet Assigned Numbers Authority), so you can be reasonably sure that they are not used by some other service. For this article, I will assume that the firewall (firewall.zeroc.com) forwards incoming connections on

Figure 1: Glacier2 behind a Firewall



ports 4063 and 4064 to the same ports on the internal machine glacier2.zeroc.com, which runs Glacier2. (The internal machine need not run Glacier2 on these ports but, seeing that they are reserved for Glacier2, you might as well use them.)

Running Glacier2

Glacier2 is provided as the command `glacier2router` in the Ice distribution. The simplest way to run Glacier2 is as follows:

```
glacier2router --Glacier2.Client.Endpoints="tcp -h
 glacier2.zeroc.com -p 4063" --Glacier2.Permission
sVerifier=Glacier2/NullPermissionsVerifier
```

The property `Glacier2.Client.Endpoints` configures the port at which Glacier2 listens for client TCP requests. In this case, it listens only on the interface bound to glacier2.zeroc.com's IP address on port 4063. As I mentioned earlier, Glacier2 can also be configured to require authentication from clients. The property `Glacier2.PermissionsVerifier` determines the authentication mechanism. Glacier2 ships with a built-in null permissions verifier that allows anyone to connect—the object identity `Glacier2/NullPermissionsVerifier` selects this "allow anyone" verifier. (I will discuss other authentication options shortly.)

Running the Server

On the server side, no configuration is required at all: to use a server with Glacier2, you simply start the server with the same configuration as you would without Glacier2.

Running the Client

For the client, we need to make minor source code changes to allow the client to use Glacier2. Specifically, clients must establish a session with Glacier2 to have their requests forwarded to servers. On start-up, the client needs to execute the following code:

```
// C++
Ice::RouterPrx r =
    communicator()->getDefaultRouter();
Glacier2::RouterPrx router =
    Glacier2::RouterPrx::checkedCast(r);
Glacier2::SessionPrx session;
try
{
    router->createSession("", "");
}
catch(const Ice::Exception& ex)
{
    cerr << "Cannot create session: " << ex
        << endl;
}
```

The call to `createSession` expects a user name and password. Because (for the moment), we are using the null permissions verifier, any user name and password will do, so the code passes empty strings. This code establishes the session that allows the client to communicate with the server via Glacier2.

Before the client terminates, it should destroy the session again:

```
//C++
try
{
    router->destroySession();
}
catch(const Ice::ConnectionLostException&)
{
    // Expected: Glacier2 destroyed the session.
}
catch(const Ice::Exception& ex)
{
    cerr << "Cannot destroy session: " << ex
        << endl;
}
```

You must catch `ConnectionLostException` when calling `destroySession` because Glacier2 closes the connection in response to this call, causing the exception to be raised in the client.

You can also make Glacier2 destroy sessions that have been idle for a while, by setting the property `Glacier2.SessionTimeout` to the idle time in seconds. It is strongly recommended to set this property to ensure correct cleanup in the event of a client crash. Regardless, it is a good idea to explicitly destroy the session to ensure timely clean-up of resources inside Glacier2. (And, if you do not configure a session timeout, sessions last indefinitely.)

The above code is all that is necessary to make a client cooperate with Glacier2. You can bundle this code into utility functions and then reuse it in your clients such that they automatically use Glacier2. For example, you can create a simple class that, in its constructor, establishes the session (and prompts the user for a user name and password, if appropriate) and in its destructor destroys the session. Client code changes are then limited to simply instantiating the class. As a refinement, the constructor of the class can check whether `getDefaultRouter` returns a null proxy; if so, the client is not configured to use Glacier2 and the constructor simply returns without establishing a session.

This allows you to use the same binary client with and without Glacier2—you can control whether the client communicates with its servers via Glacier2 by adjusting the client's configuration.

To get the client to use Glacier2, we need a minimum of configuration:

```
# Client configuration
Ice.Default.Router=Glacier2/router:tcp -h firewall
.zeroc.com -p 4063
Ice.ACM.Client=0
Ice.RetryIntervals=-1
```

The property `Ice.Default.Router` configures a default router for the client. Setting this property has the effect that all client requests are sent to the object specified by that property, instead of being sent to the endpoint that is inside the proxy that a client uses to make an invocation. In effect, the property says "send all invocations to the specified object, instead of sending them as you normally would." The host and port for this property must point at the firewall, which port-forwards all traffic to Glacier2 to the host and port set by Glacier2's `Glacier2.Client.Endpoints` property.

Glacier2 also requires the client to disable automatic connection management (ACM). This is necessary because, once a client drops its connection to Glacier2, Glacier2 automatically destroys the client's session. Setting the property `Ice.ACM.Client` to zero prevents the client from closing its connection to Glacier2 due to idle periods and so having its session disappear unexpectedly.

Finally, retries do not make sense if a client communicates with a server via Glacier2 because Glacier2 will retry failed requests automatically on behalf of the client. To disable retries, we set `Ice.RetryIntervals` to a negative value.

This is all that is needed to get off the ground, at least for this simple scenario: run Glacier2, add the preceding few lines of code to the client, run the server, and run the client with these three configuration items.

If you have problems getting things to work, it will almost certainly be due to incorrect endpoint configuration. In particular, the client's `Ice.Default.Router` setting must point at the firewall and the firewall must forward to the host and port defined with `Glacier2.Client.Endpoints`. You can set `Ice.Trace.Network=1` for Glacier2 and the client to check whether connections are made to the correct address and port.

Better Authentication

You can force clients to authenticate themselves with a user name and password when they create a Glacier2 session by leaving `Glacier2.PermissionsVerifier` undefined, and instead setting the property `Glacier2.CryptPasswords` to the path name of a password file. Doing this activates a built-in permissions verifier that uses the Unix `crypt` algorithm to authenticate clients.

The password file must contain pairs of user name and encrypted passwords, one per line. The client passes the user name and (unencrypted) password to `createSession`, and Glacier2 allows access only if the supplied password encrypts to the same string that is stored in the password file. Note that if you use this mechanism, you should restrict client access to SSL, otherwise the password will be sent in clear text over the wire.

Using SSL

For security-sensitive applications, you will probably want to ensure that no-one can eavesdrop on the traffic between clients and Glacier2 and use an SSL connection instead of TCP. To run Glacier2 with SSL and disable TCP, you need to set a few additional properties:

```
# Glacier2 config for SSL
Glacier2.Client.Endpoints=ssl -h glacier2.zeroc.
com -p 4064
Glacier2.PermissionsVerifier=Glacier2/NullPermissi
onsVerifier
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=certs
IceSSL.CertAuthFile=cacert.pem
IceSSL.CertFile=s_rsa1024_pub.pem
IceSSL.KeyFile=s_rsa1024_priv.pem
IceSSL.VerifyPeer=0
```

Note that Glacier2 now uses an SSL endpoint. The remaining properties specify that the Ice run time should load the SSL plug-in (`Ice.Plugin.IceSSL`) and configure the directory and files that provide the plug-in with the relevant certificate and key information. We are still using the null permissions verifier, so any client can connect, but only via SSL. Because the client is still authenticated via user name and password, it need not provide its own SSL credentials: Glacier2 sets `IceSSL.VerifyPeer` to zero to accept such anonymous connections. (This example uses the certifcates that accompany the Ice distribution. For a real-world deployment, you would generate your own CA certificate and a unique certificate for the Glacier2 router. See the Ice Manual for more details on how to configure the SSL plug-in and how to generate certificates.)

As before, no changes are required for the server—the server can provide either TCP or SSL endpoints, and Glacier2 will forward client requests to the server as appropriate.

The client must be configured as follows:

```
# Client configuration
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=certs
IceSSL.CertAuthFile=cacert.pem
IceSSL.TrustOnly=CN="Server"
```

The `IceSSL.TrustOnly` rule tells the client to connect only to servers that have the common name `Server`. In a real-world deployment, you would use a unique common name for your Glacier2 router, and use that common name instead.

With this configuration, clients communicate with Glacier2 only via SSL, and Glacier2 forwards requests to back-end servers using whatever endpoints (TCP or SSL) are provided by these servers.

Using SSL Connection Credentials

You can use SSL in combination with user name and password authentication exactly as with TCP: leave `Glacier2.PermissionsVerifier` undefined, and instead set `Glacier2.CryptPasswords` to the path name of the password file. With that configuration, because SSL communications are encrypted, the client's password is no longer sent in plain text over the wire when the client calls `createSession`.

An alternative way to authenticate clients is to use the credentials that are established for the client's SSL connection. In that case, the client does not need to supply a password—instead, the client calls `createSessionFromSecureConnection`, which does not require arguments:

```
// C++
try
{
    router->createSessionFromSecureConnection();
}
catch(const Ice::Exception& ex)
{
    cerr << "Cannot create session: " << ex
        << endl;
}
```

For this to work, Glacier2 must be configured slightly differently: instead of setting `Glacier2.PermissionsVerifier` or `Glacier2.CryptPasswords`, leave these properties undefined and set `Glacier2.SSLPermissionsVerifier` instead:

```
# Glacier2 config for SSL
Glacier2.SSLPermissionsVerifier=Glacier2/NullSSLP
ermissionsVerifier
# Other settings as before...
```

The value `Glacier2/NullSSLPermissionsVerifier` allows any client to connect, provided that the SSL connection could be established. If you want to restrict access to specific clients, you need to install a custom verifier.

Custom Verifiers

You can set `Glacier2.PermissionsVerifier` to the proxy of an arbitrary Ice object that you provide in any server that is reachable by Glacier2. The target object must implement the `Glacier2::PermissionsVerifier` interface, which contains a `checkPermissions` operation. To verify a client's password, Glacier2 calls your `checkPermissions` operation to decide whether the client should be authorized.

Similarly, you can set `Glacier2.SSLPermissionsVerifier` to the proxy of an Ice object that implements the `Glacier2::SSLPermissionsVerifier` interface, which contains an `authorize` operation that Glacier2 invokes when the client calls `createSessionFromSecureConnection`.

This allows you to implement arbitrary authorization policies, typically by delegating the decision to an authorization mechanism that you have already in place.

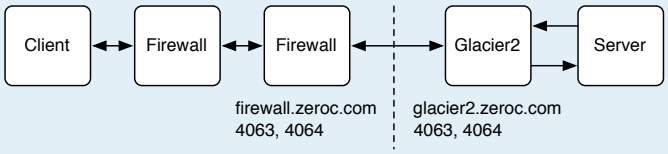
Using Callbacks

With the setup we have seen so far, clients can reach servers through the firewall, but servers cannot necessarily reach clients. Doing this is necessary if a client passes a proxy to a callback object to a server. In that case, the client is both client and server and, when the server calls back into the client, they momentarily reverse roles: the server acts as the client, and the client acts as the server.

There is nothing wrong with this as such: if the client is not behind a firewall of its own, the server will simply open a connection to the client and invoke the callback via that connection. However, chances are that the client will be behind its own firewall, with that firewall disallowing incoming connections (see Figure 1).

The solution for this problem is for the server to send the callback to Glacier2, which forwards the callback to the correct client via the already-existing connection that was established by the client. That way, the server can reach the client even if the client is behind a firewall that disallows incoming connections, as shown in Figure 2.

Figure 2: Bi-Directional Communication with Glacier2 for Callbacks



To make this setup work, no code or configuration changes are necessary in the server. However, we need to add one additional property setting to Glacier2's configuration:

```
# Glacier2 config
Glacier2.Server.Endpoints=tcp -h glacier2.zeroc.
com
# Other settings as before...
```

Setting `Glacier2.Server.Endpoints` enables an endpoint in Glacier2 that servers use when they invoke a callback on a client. (Note that you need not specify a port number for this property.) The endpoint you specify here must be accessible on the internal network, so the back-end servers can connect to it, and should not be accessible from the external network, to prevent malicious clients from flooding Glacier2's server endpoint with requests.

The million-dollar question is: how does it happen that servers connect to Glacier2's server endpoint when they invoke a callback,

instead of attempting to open a separate connection directly to the client? The answer involves two separate things on the client side. The first is that the client must have an additional property setting:

```
# Client config
CallbackAdapter.Router=Glacier2/router:ssl -h
firewall.zeroc.com -p 4064
# Other settings as before...
```

Note the setting of `CallbackAdapter.Router`. (This assumes that the client's object adapter that provides the callback object has the name `CallbackAdapter`.)

The property configures the client's object adapter with a router, and the setting of that property must point at the firewall. Setting this property has the effect that proxies for callback objects that the client creates contain the server endpoint of the Glacier2 router that is used by back-end servers (instead of the endpoint at which the client's object adapter listens). This explains how, when the server invokes a callback, it ends up connecting to Glacier2 instead: the client-side run time notices the property setting, asks Glacier2 for the endpoint that Glacier2 provides to servers for callbacks, and puts that server endpoint (which is on the internal network) into the callback proxy. Therefore, the back-end server connects to Glacier2's server endpoint when it invokes the callback.

The second part of the answer deals with how Glacier2 can ensure that callbacks for different clients actually go to the correct client: because each server has only a single connection to Glacier2, but may need to send callbacks to different clients, the identity of the target client is no longer implicit in the server's connection to Glacier2. Instead, the client must provide an identifier that Glacier2 can use to de-multiplex callbacks from back-end servers in order to forward them to the correct client.

Glacier2 does this by assigning a unique identifier to each client. In turn, the client is expected to provide that identifier in the `category` part of the object identity for its callback objects. For example, suppose the client provides callback objects of interface `Callback` to a number of back-end servers. The client must contact Glacier2 once, to obtain the unique category Glacier2 has assigned to the client, and then use that category in the object identity of its callback objects:

```
// C++
// Get category from Glacier2.
string myCategory =
    router->getCategoryForClient();

// Use that category for all callback objects.
Identity id;
id.category = myCategory;

// Create two callback objects with name cb1
// and cb2.
id.name = "cb1";
adapter->add(new CallbackI(), id);
id.name = "cb2";
adapter->add(new CallbackI(), id);
```

Instead of explicitly assigning the category in-line, as shown by the preceding code, I suggest that you bundle the object identity creation as a `makeId` method into the same helper class I suggested earlier. The helper class, if the client is configured with a router, automatically assigns the category and, if no router is configured, leaves the category empty. That way, the same binary client can be used with and without Glacier2 by simply changing its configuration. The calls to `adapter->add` then use `makeId` on the helper class:

```
// C++
// Instantiate router helper.
RouterHelperPtr rh =
       new RouterHelper(communicator());

// Create two callback objects with name cb1
// and cb2.
adapter->add(new CallbackI(), rh->makeId("cb1"));
adapter->add(new CallbackI(), rh->makeId("cb2"));
```

The helper class, in outline, looks something like this (methods are in-line only for brevity):

```
// C++
class RouterHelper : public IceUtil::Shared
{
public:
    RouterHelper(
        const CommunicatorPtr& communicator)
    {
        Ice::RouterPrx r =
            communicator->getDefaultRouter();
        if(r)
        {
            _router =
              Glacier2::RouterPrx::checkedCast(r);
            if(!_router)
            {
                throw
                    "Wrong interface for router";
            }
            string name;
            string password;
            // Initialize name and
            // password here...

            _router->createSession(
                name, password);
            _category =
                _router->getCategoryForClient();
        }
    }
```

```
~RouterHelper()
    {
        try
        {
            if(_router)
            {
                _router->destroySession();
            }
        }
        catch(...)
        {
        }
    }

    Identity makeId(const string& name)
    {
        Identity id;
        id.name = name;
        id.category = _category;
        return id;
    }

private:
    Glacier2::RouterPrx _router;
    string _category;
};
typedef IceUtil::Handle<RouterHelper>
    RouterHelperPtr;
```

You can easily modify this helper class to suit your own needs, for example, to dynamically select the correct permissions verifier for SSL, or to delegate authentication to the appropriate mechanism.

Summary

Glacier2 makes it very easy to provide secure access to Ice servers that sit behind a firewall. Once you know Glacier2, you can make a new server available in just a few minutes. The coding effort required to make clients cooperate with Glacier2 is truly minimal: you only need to write a few lines of code once and then can re-use that code in all your clients. Moreover, it is trivial to write the helper code such that it works with and without a router; by doing this, you can switch an existing client from non-routed to routed operation simply by changing the client's configuration.

If you want to experiment with Glacier2, I suggest you start with the demo that is provided in the `demo/Glacier2/callback` directory in the Ice distribution. The demo also illustrates how to connect a custom verifier to Glacier2, and how to use explicit session management. In addition, I suggest that you take a look at the Glacier2 chapter in the Ice Manual, which provides more information on advanced features, such as fine-grained access control, integration with IceGrid, and other topics. And, as always, if you would like to discuss the topic of this article, you can get in touch with us in our developer forums.

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/forums/ 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 do I use IceSSL with Ruby/Python/PHP?

Ice for Ruby, Ice for Python, and Ice for PHP are based on Ice for C++. Therefore, configuring IceSSL for these languages works the same way as with Ice for C++. With Ice for Ruby and Ice for Python, configuring IceSSL is straightforward. The configuration files for some of the demos that accompany the distributions are examples of how to do this. For example, if you inspect the `config.client` file for the Ruby or Python `demo/Ice/hello`, you will find the following configuration items for IceSSL:

```
# config.client
# ...
#
# Security Tracing
#
# 0 = no security tracing
# 1 = trace messages
#
#IceSSL.Trace.Security=1

#
# SSL Configuration
#
Ice.Plugin.IceSSL=IceSSL:createIceSSL
IceSSL.DefaultDir=../../../certs
IceSSL.CertAuthFile=cacert.pem
IceSSL.CertFile=c_rsa1024_pub.pem
IceSSL.KeyFile=c_rsa1024_priv.pem
```

The `Ice.Plugin.IceSSL` property tells the Ice run time to load the IceSSL plug-in, and the properties prefixed by `IceSSL` configure the plug-in itself. For more information on these properties, please see the Ice Manual. (Note that, under Windows, Ice for Ruby has an OpenSSL compatibility issue. Please see the `INSTALL.WINDOWS` file that accompanies the distribution for details.)

The procedure for configuring Ice for PHP is the same: configure the IceSSL plug-in as you would for any C++ application, and inform the Ice-for-PHP plug-in to use this configuration. However, because the plug-in is loaded by Apache, it can be a little tricky to get things working. If you are loading the Ice-for-PHP plug-in dynamically, the IceSSL and OpenSSL shared libraries must be accessible to Apache. In addition, you must configure IceSSL. One way to do this is to put the property settings (such as the above) into a configuration file that is accessible to Apache, and then add an `ice.config` directive to the `php.ini` file. For example, assuming the configuration file is located in `/etc/config-ice.php`, and you are adding to the default PHP profile:

```
# php.ini
# ...
ice.config=/etc/config-ice.php
```

Another method is to add an `ice.options` directive to the `php.ini` file. On the up-side, this avoids external configuration files but, on the down-side, makes the `php.ini` file more verbose.