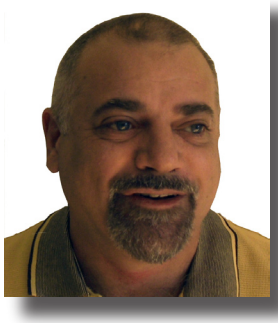
Connections

ZeroC's Newsletter for the Ice Community

Issue Number 18, October 2006



Abstraction Layers

A fundamental technique of dealing with complexity in computing is to provide abstractions. What makes abstractions work is that they throw away irrelevant information and retain only information that is necessary for a particular purpose. By throwing away information, abstractions reduce complexity to manageable levels.

Ice is an abstraction that presents a simplified picture of a complex network infrastructure: as a programmer, you can call an operation on a remote object and, in a nutshell, Ice lets you know whether the operation worked or not. Underneath the covers, doing this requires a lot of complex activity that you can safely ignore—the whole idea of Ice is that it lets you get on with the job of building your application instead of worrying about the network.

Abstractions are everywhere in computing. In particular, *layers* of abstractions are everywhere. We not only create abstractions, we also create abstractions on top of other abstractions. For example, at the bottom-most layer, we have hardware. On top of that, we have device drivers and an operating system. On top of the OS, we have system libraries that provide various services. Typically, these libraries are themselves arranged into layers, with one library calling on the services of another. And on top of the libraries, we have applications that typically are themselves structured as layers of abstractions. Layering of abstractions is an immensely powerful and useful idea—without it, computing as we know it could not exist.

Layered abstractions are not without problems. For example, we all probably had moments where we were confounded by some mysterious and inexplicable failure of code, only to ultimately find out that the failure was caused by some lower layer. Diagnosing such problems can be very difficult: by definition, abstractions throw away information (otherwise they would not be abstractions) but, occasionally, it is precisely the information that is thrown away that we need to find out why things do not work.

It is interesting to note that, the lower in the abstraction hierarchy a problem occurs, the less tolerant we are of the problem. For example, bugs in compilers generally elicit extremely vitriolic reactions from developers. Similarly, you might remember the Intel Pentium floating point bug. The problem caused a huge public outcry, even though the bug affected very few people. What makes us so intolerant of lower-level bugs is that they bite indiscriminately and, therefore, can cause a lot of consequential damage.

If you develop applications with Ice, you will probably think of Ice as something quite low in your hierarchy of abstractions, probably somewhere just above the OS level. If your application does not communicate the way you think it should, you would like to know why. And, if you can't easily find out why things do not work, you are probably upset. (Believe me, I know what that feels like, because *I* get upset every time one of *my* Ice applications does not work and I cannot easily find out why.)

Sometimes your problem will be caused by something you have done wrong in your code. To help you find out what that is, you have quite a few resources to turn to. For one, there is *Connections*, which you are reading right now. I particularly recommend the FAQ section, which provides answers to problems that fellow developers *really* have run into. (No, we don't invent these questions; they come from our developer community.) There is also the extensive Ice manual, which, apart from telling you how things work, provides many tips for how to stay out of trouble and how to debug things. And there is our developer forum, where you get help from us and other members of the Ice community.

And sometimes, your problem will be caused by a bug in Ice. (Yes, we are not perfect and make mistakes like everyone else.) We care about our developers, and we apologize for the inconvenience a bug causes you. When you do find a bug, please let us know about it. We will do our best to solve your problem, and to solve it *quickly*. At ZeroC, we take pride in the quality of our software, and we are indebted to you when you find a bug in Ice and tell us about it: every time that happens, Ice becomes a better abstraction.

Michi Henning
Chief Scientist

Issue Features

Session Management with IceGrid

This article discusses the use of an interposed IceGrid session to prevent malicious clients from monopolizing a grid.

Write Once, Read Everywhere

Michi Henning describes how to use slice2docbook.

Contents

Session Management with IceGrid

Matthew Newhook, Senior Software Engineer

Introduction

The previous article in this series (see Issue 17 of *Connections*) showed how to secure a grid against unauthorized access. However, the application did nothing to protect itself from malicious or otherwise broken clients. This article demonstrates how we can add such protection.

The MP3 encoder application we developed so far had the following goals:

- The application must not run more encoders to run in parallel than the number of CPUs (or some factor thereof).
- Each factory allows only one file to be encoded at a time.
- Each factory is reserved for the exclusive use of a single client until the client no longer uses that factory.

We satisfied these goals in the following way:

- We used the IceGrid session allocation mechanism to allocate factories to clients.
- On each encoding machine, we deployed a limited number of encoder factory services according to the number of CPUs on that machine.

Each client could allocate only one encoder factory and therefore encode only one file at time. However, we assume that we have an entire server farm, and we would like to keep the servers as busy as possible, so we would like to allow clients to allocate more than one factory, so that clients can encode several files in parallel. However, a potential problem of this strategy is that a client, due to bandwidth and processing limitations, may not be able to fully utilize the encoders it has allocated. Therefore, we want to allow clients to allocate more than one encoder, but not more encoders than the clients each can reasonably use. Also, we do not want a malicious or broken client to allocate all of the available encoders because that would deny service to other clients. A simple strategy for dealing with this is to limit the number of encoders that a client can allocate to some (small) fixed number. However, IceGrid cannot enforce this limitation, so we need to take of it ourselves.

Implementation

First, we will deal with how to prevent a client from allocating more than a fair share of resources. Initially, we will limit the client to allocating only one encoder factory per session. Once we have accomplished this, it is fairly simple to modify things such that clients can allocate several encoders, up to some specific limit.

A simple approach to solving the problem would be to introduce a new Glacier2 session manager object that performs the session allocation on behalf of the client. For example:

```
// Slice
interface EncoderSessionManager extends
    Glacier2::SessionManager
{
    Ripper::EncoderFactory* allocate();
};
```

When a client creates this object, the implementation could establish an IceGrid session and release that session again when the object is destroyed. This strategy would certainly work but has the drawback that it requires code changes in the client. However, we can also create an implementation that does not require any client code changes: we can create an implementation of the `IceGrid::Session` interface that, while imposing additional restrictions on the client, delegates parts of its implementation to an actual IceGrid session.

Glacier2 creates sessions by delegating the creation to a session manager. Recall that there are two supported methods for user authentication with Glacier2. The first uses user name and password authentication, and the second uses the credentials associated with an SSL connection. There are two session manager interfaces, one for each authentication method:

```
// Slice
interface SessionManager
{
    Session* create(string userId,
        SessionControl* control)
        throws CannotCreateSessionException;
};

interface SSLSessionManager
{
    Session* create(SSLInfo info,
        SessionControl* control)
        throws CannotCreateSessionException;
};
```

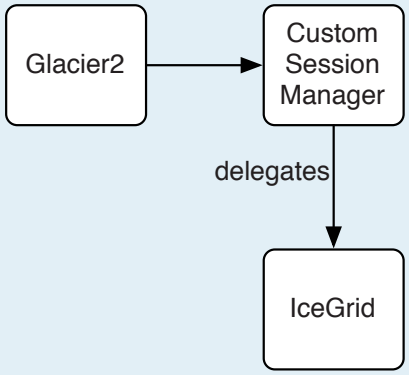
These are configured with the properties `Glacier2.SessionManager` and `Glacier2.SSLSessionManager`, respectively. The previous application configuration for Glacier2 was something like the following:

```
// IceGrid descriptor
<server-instance template="Glacier2"
 client-endpoints="ssl -h 192.168.1.102 -p 10005"
 server-endpoints="tcp"
 session-timeout="30">
  <properties>
    <property name="Glacier2.SessionManager"
     value="EncoderIceGrid/SessionManager"/>
    ...
  </properties>
</server-instance>
```

This descriptor tells Glacier2 to use the IceGrid session manager to create its sessions. It is important to note that access to the session manager should normally be restricted to authorized users. Glacier2 invokes the session manager only after it has successfully validated the client's credentials, but you must also prevent a malicious client from directly invoking operations on the session manager. You can do this in a number of ways, such as by forcing all interaction with backend servers to go through Glacier2, or by using SSL with sufficient access controls.

What we will do is configure Glacier2 to use our custom session manager, and configure our custom session manager to delegate to the IceGrid session manager, that is, we interpose the custom session manager between Glacier2 and IceGrid and enforce the limit on the number of files a client can encode in the interposed implementation.

Figure 1: Interposed Session Manager



Let's start with the implementation, beginning with the session manager. The class definition is as follows:

```
// C++
class SessionManagerI :
    public Glacier2::SessionManager
{
public:
    SessionManagerI(
        const Glacier2::SessionManagerPrx&);
    ~SessionManagerI();

    virtual Glacier2::SessionPrx
    create(const std::string&,
           const Glacier2::SessionControlPrx&,
           const Current&);

private:

    const Glacier2::SessionManagerPrx _manager;
};
```

The implementation of the constructor and destructor are straightforward; see the accompanying source code for details. We'll move on to the implementation of session creation:

```
// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Current& current)
{
    IceGrid::SessionPrx session =
        IceGrid::SessionPrx::uncheckedCast(
            _manager->create(userId, control));
    Glacier2::SessionPrx interposed =
        Glacier2::SessionPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new SessionI(session)));
    return interposed;
}
```

We allocate a session from the IceGrid session manager, allocate our custom session manager implementation, and then return a proxy to the custom session manager.

Now we move onto the implementation of the custom session manager. The Slice definition for the interface is shown below:

```
// Slice
interface Session extends Glacier2::Session
{
    idempotent void keepAlive();
    ["ami", "amd"] Object*  allocateObjectById(
        Ice::Identity id)
        throws ObjectNotRegisteredException,
            AllocationException;
    ["ami", "amd"] Object* allocateObjectByType(
        string type)
        throws AllocationException;
    void releaseObject(Ice::Identity id)
        throws ObjectNotRegisteredException,
            AllocationException;
    idempotent void setAllocationTimeout(
        int timeout);
};
```

As a first step, we'll delegate all calls directly to the IceGrid session to ensure that all of the plumbing is hooked up correctly. Here is a partial implementation:

```
// C++
class SessionI : public IceGrid::Session
{
public:

    SessionI(const IceGrid::SessionPrx& session) :
        _session(session)
    {
    }

    ~SessionI()
    {
    }
```

```
    virtual void
    keepAlive(const Current&)
    {
        _session->keepAlive();
    }


    virtual void
    releaseObject(const Identity& id,
        const Current&)
    {
        _session->releaseObject(id);
    }

    virtual void
    setAllocationTimeout(int timeout,
        const Current&)
    {
        _session->setAllocationTimeout(timeout);
    }

    virtual void
    destroy(const Current& current)
    {
        current.adapter->remove(current.id);
        _session->destroy();
    }

    // ...

private:

    const IceGrid::SessionPrx _session;
};
```

Next we'll deal with the allocation methods, which are a bit trickier because the above interface uses asynchronous method dispatch (AMD). We could have used synchronous operations instead, but session allocation is likely to be slow because it typically will have to wait for an encoder object to become available. With a synchronous implementation, the server would consume a thread from the server-side dispatch pool for every client that is waiting for an encoder to become available. However, we expect a session server to support many clients, and we expect many clients to be waiting, so the number of clients that could be waiting at a time would be limited by the number of threads that the server has in its pool. Because threads are expensive in terms of memory consumption, this does not scale all that well and, by using AMD, we can have an unlimited number of clients waiting for an encoder without consuming a separate thread for each client for the duration of the wait. (If you are not familiar with the details of asynchronous method invocation and dispatch, have a look at *Asynchronous Programming* in Issue 4 of *Connections*.)

Let's look at the implementation of these methods:

```
// C++
virtual void
allocateObjectById_async(
    const AMD_Session_allocateObjectByIdPtr&
        response,
    const Identity& id, const Current&)
{
    _session->allocateObjectById_async(
      new AMI_Session_allocateObjectByIdI(
        response), id);
}

virtual void
allocateObjectByType_async(
    const AMD_Session_allocateObjectByTypePtr&
         response,
    const string& type, const Current&)
{
    _session->allocateObjectByType_async(
        new AMI_Session_allocateObjectByTypeI(
            response), type);
}
```

The operations use the technique of AMD/AMI chaining, which releases the server-side invocation thread as soon as the AMI invocation is sent to the IceGrid session.

The implementation of the AMI callback objects is trivial and simply notifies the caller of the completion of an invocation, either normally, or via an exception:

```
// C++
class AMI_Session_allocateObjectByIdI :
    public AMI_Session_allocateObjectById
{
public:

    AMI_Session_allocateObjectByIdI(
      const AMD_Session_allocateObjectByIdPtr&
          response) :
        _response(response)
    {
    }

    virtual void
    ice_response(const ObjectPrx& obj)
    {
        _response->ice_response(obj);
    }

    virtual void
    ice_exception(const Exception& e)
    {
        _response->ice_exception(e);
    }

private:

    const AMD_Session_allocateObjectByIdPtr
        _response;
};
```

The implementation of `AMI_Session_allocateObjectByTypeI` is essentially the same so I have omitted it—see the source code for full details.

Now we can turn our attention to implementing the session manager as an IceBox service:

```
// C++
class SessionManagerServiceI :
    public IceBox::Service
{
public:
    virtual void
    start(const string& name,
        const CommunicatorPtr& communicator,
        const StringSeq& args)
    {
        Glacier2::SessionManagerPrx manager =
            Glacier2::SessionManagerPrx::
            uncheckedCast(
              communicator->stringToProxy(
                communicator->getProperties()->
                  getProperty("SessionManager")));
        assert(manager);

        _adapter = communicator->
            createObjectAdapter(
                name + "-SessionManager");
        _adapter->add(
            new SessionManagerI(manager ),
            communicator->stringToIdentity(
                name + "-SessionManager"));
        _adapter->activate();
    }

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

private:

    ObjectAdapterPtr _adapter;
};
```

The service binds to the IceGrid session manager that is configured in the `SessionManager` property, creates an object adapter for the session manager, and then creates the session manager implementation itself. Note that we use the session manager instance name to create unique object adapter and object instance names so that we can have multiple session managers if necessary.

Now we can discuss the deployment descriptor. First, we'll write a template for the `SessionManager` server.

```
// IceGrid deployment descriptor
<server-template id="SessionManager">
  <parameter name="instance-name"/>
  <parameter name="host"/>
  <icebox id="${instance-name}" exe="icebox"
   activation="on-demand">
    <property name="SessionManager"
     value="EncoderIceGrid/SessionManager"/>
    <service name="${instance-name}"
        entry="SessionManagerService:create">
        <adapter
         name="${instance-name}-SessionManager"
         endpoints="tcp -h ${host}">
          <object
          identity="${instance-name}-SessionManager"
           type="::Glacier2::SessionManager"/>
        </adapter>
    </service>
  </icebox>
</server-template>
```

We can deploy the template on the `localhost` node:

```
// IceGrid deployment descriptor
<node name="localhost">
  <server-instance
    template="SessionManager"
    instance-name="localhost-manager"
    host="localhost"/>
   ...
</node>
```

We also need to change the Glacier2 descriptor to use the correct session manager.

```
// IceGrid deployment descriptor
<property name="Glacier2.SessionManager"
 value="localhost-manager-SessionManager"/>
```

Ok, now let's try it to make sure all of the plumbing works. After starting up IceGrid and deploying the application, we run the client. However, we'll get the following error:

```
$ ./client testcase.wav
./client: Outgoing.cpp:368: Ice::ObjectNotExistExc
eption:
object does not exist:
identity: `66B8126A-5212-4963-BDD2-F04507D49385'
facet:
operation: allocateObjectByType
```

A good way to get to the bottom of this is to enable Glacier2 filter tracing and inspect the logs. You can enable tracing by modifying the `properties` section of the Glacier2 deployment:

```
// IceGrid deployment descriptor
<property name="Glacier2.Client.Trace.Reject"
  value="1"/>
```

Furthermore, let's add some tracing to our session manager implementation:

```
// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Current& current)
{
    IceGrid::SessionPrx session =
        IceGrid::SessionPrx::uncheckedCast(
            _manager->create(userId, control));
```

```
    CommunicatorPtr communicator =
        current.adapter->getCommunicator();
    Trace trace(communicator->getLogger(),
        "SessionManagerI");
    trace << " allocated session: "
          << communicator->proxyToString(session);
    Glacier2::SessionPrx interposed =
    Glacier2::SessionPrx::uncheckedCast(
        current.adapter->addWithUUID(
            new SessionI(session)));
    trace << " interposed: " << communicator->
        proxyToString(interposed);
    return interposed;
}
```

After updating the deployment, restarting the Glacier2 router, and re-running the client, we find the following in the logs:

```
[ localhost-manager: SessionManagerI:  allocated
session: EncoderIceGrid/6926E105-4466-4333-818E-
D374B9319377 -t:tcp -h 192.168.1.5 -p 51341
interposed: 66B8126A-5212-4963-BDD2-F04507D49385 -
t @ localhost-manager.localhost-manager.localhost-
manager-SessionManager ]
[ glacier2router: Glacier2: rejecting request:
identity filter identity: 66B8126A-5212-4963-BDD2-
F04507D49385 ]
```

The trace indicates that Glacier2 does not permit access to the interposed session because the configured filters deny access to this object. At this point, I want to look at Glacier2 filtering in more detail because the topic is important for our implementation.

The primary purpose of Glacier2 filtering is to ensure that Glacier2 clients do not gain access to unintended objects. Glacier2 supports the following types of filters. The different filter types progressively move from coarse-grained to fined-grained levels of access control.

- Address Filters: Permit or deny client side access to a given set of hosts and ports.
- Category Filters: Configure the set of Ice identity categories that clients can access.
- Identity Filters: Configure access to a given set of identities.
- Adapter Filters: Configure access to all objects accessed via an indirect proxy with a given object adapter.

The default for all of these filters is to permit access to all objects. The properties can be configured statically (using the `Glacier2.Filter` family of properties), and they can also be altered at run time using the `Glacier2::SessionControl` interface.

The session control and other interfaces related to dynamic control of the Glacier2 filtering are shown below:

```
// Slice
module Glacier2
{
interface StringSet
{
    idempotent void add(Ice::StringSeq additions);
    idempotent void remove(
        Ice::StringSeq deletions);
    idempotent Ice::StringSeq get();
};

interface IdentitySet
{
    idempotent void add(
        Ice::IdentitySeq additions);
    idempotent void remove(
        Ice::IdentitySeq deletions);
    idempotent Ice::IdentitySeq get();
};

interface SessionControl
{
    StringSet* categories();
    StringSet* adapterIds();
    IdentitySet* identities();
    void destroy();
};
};
```

The `SessionControl` interface gives access to the category, adapter, and identity filters. Calling `destroy` causes the router to destroy the client's session, which eventually results in `destroy` being called on the IceGrid session. Calling `identities`, `adapterIds`, or `categories` returns a proxy to an object that can be used to add, remove, and retrieve the current set of filter configuration values.

The Glacier2 router immediately terminates a client's session if the client attempts to use a proxy that is rejected by an address filter. The Ice run time in the client responds by raising `ConnectionLostException` to the application. If a client attempts to use a proxy that is rejected by a category, identity, or adapter identifier filter, the router raises `ObjectNotExistException`.

IceGrid itself configures the Glacier2 filtering using the `Glacier2::SessionControl` object. By default, IceGrid configures Glacier2 to only permit access to the `IceGrid::Session` and `IceGrid::Query` objects. IceGrid updates the identity filters when a client allocates or releases an object via its IceGrid session. When a client allocates a server, IceGrid adds adapter identity filters for the server's indirect adapters and removes the filters again once the server is released.

Given the above description, we can now see why the client cannot use the interposed session object: the default filters established by IceGrid deny access to an object with the identity of the interposed session. To fix this, we can add an identity filter for the interposed object:

```
// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Current& current)
{
    IceGrid::SessionPrx session =
|       IceGrid::SessionPrx::uncheckedCast(
            _manager->create(userId, control));
    CommunicatorPtr communicator =
        current.adapter->getCommunicator();
    Trace trace(communicator->getLogger(),
        "SessionManagerI");
    trace << " allocated session: "
         << communicator->proxyToString(session);
    Glacier2::SessionPrx interposed =
        Glacier2::SessionPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new SessionI(session)));
    IdentitySeq ids;
    ids.push_back(interposed->ice_getIdentity());
    control->identities()->add(ids);
    trace << " interposed: " << communicator->
        proxyToString(interposed);
    return interposed;
}
```

When we try the client again, we find that, with this change, everything works as expected. There are a couple of problems, however. Firstly, the identity filter is never removed when the session is destroyed. Doing this isn't strictly necessary the filters are reset for each new session, and `destroy` is only called when the session is over. However, for cleanliness we'll make this change. We pass the `Glacier2::SessionControl` object to the interposed session and, when the session is destroyed, we remove the filter:

```
// C++
class SessionI : public IceGrid::Session
{
public:

    SessionI(
    const Glacier2::SessionControlPrx& control,
    const IceGrid::SessionPrx& session) :
        _control(control),
        _session(session)
    {
    }
    // . . .

    virtual void
    destroy(const Current& current)
    {
        current.adapter->remove(current.id);
        IdentitySeq ids;
        ids.push_back(current.id);
        _control->identities()->remove(ids);

        _session->destroy();
    }

private:

    const Glacier2::SessionControlPrx _control;
    const IceGrid::SessionPrx _session;
};
```

And when allocated the control must be passed to the interposed session:

```
// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Current& current)
{
    IceGrid::SessionPrx session = ...;
    Glacier2::SessionPrx interposed =
        Glacier2::SessionPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new SessionI(control, session)));
    // ...
```

Having dealt with the filter removal issue, we still have a more subtle problem: the Glacier2 filtering is too permissive. Remember that our primary goal is to stop the client from doing undesirable things, namely, allocating too many objects. When IceGrid creates a session object, it adds an identity filter to Glacier2 to permit access to the session. If a client somehow obtains a proxy to the IceGrid session object, it could call this object directly, thus bypassing whatever additional restrictions we place in the interposed session object. Therefore we must remove access to the IceGrid session object from the identity filter. We can do this easily enough by changing the session allocation method:

```
// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Current& current)
{
    IceGrid::SessionPrx session =
        IceGrid::SessionPrx::uncheckedCast(
            _manager->create(userId, control));
    CommunicatorPtr communicator =
        current.adapter->getCommunicator();
    Trace trace(communicator->getLogger(),
        "SessionManagerI");
    trace << " allocated session: "
        << communicator->proxyToString(session);
    Glacier2::SessionPrx interposed =
        Glacier2::SessionPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new SessionI(control, session)));
    Glacier2::IdentitySetPrx identities =
        control->identities();
    IdentitySeq ids;
    ids.push_back(interposed->ice_getIdentity());
    identities->add(ids);
```

```
    ids.clear();
    ids.push_back(session->ice_getIdentity());
    identities->remove(ids);
    trace << " interposed: "    << communicator->
        proxyToString(interposed);
    return interposed;
}
```

Adding whatever restrictions we want to place on the client now is quite straightforward. For example, let's say that we want to restrict the client to allocating at most 3 objects. To do this, we'll add a counter to the session implementation, increment the counter when each object is allocated, and decrement the counter when the object is released. The only tricky part is determining when to increment the counter. Recall that the current implementation is as follows:

```
// C++
void
SessionI::allocateObjectById_async(
    const AMD_Session_allocateObjectByIdPtr&
        response,
    const Identity& id, const Current&)
{
    _session->allocateObjectById_async(
        new AMI_Session_allocateObjectByIdI(
            response), id);
}
```

Should we increment the counter here or when the AMI callback gets the result? If we only increment the counter when the object is actually allocated by the session then, once the counter reaches the limit, we would have to immediately release the object again. But that is awkward (and needlessly allocates an object that won't be used.) Instead, it is better to increment the counter when the `allocateObjectById` method is called, and decrement the counter again if the allocation fails or when the object is later released.

Here is the implementation for `allocateObjectById`. Note that we avoid holding the lock that protects the counter during the remote calls on the session in `allocateObjectById` and `releaseObject`. (In general, it is best to avoid holding locks while making remote invocations in order to avoid deadlocks—see Bernard's articles in Issue 4 and Issue 5 of *Connections*.) The implementation for `allocateObjectByType` is identical and not shown.

```
// C++
class SessionI : public IceGrid::Session,
    public IceUtil::Mutex
{
public:

    SessionI(
        const Glacier2::SessionControlPrx&
            control,
        const IceGrid::SessionPrx& session) :
    _control(control),
    _session(session),
    _allocated(0)
    { }

    virtual void
    allocateObjectById_async(
        const AMD_Session_allocateObjectByIdPtr&
            response,
        const Identity& id, const Current&)
    {
        {
            Lock lock(*this);
            if(_allocated > 3)
            {
                throw AllocationException(
                    "allocated too many objects");
            }
            ++_allocated;
        }
        _session->allocateObjectById_async(
            new AMI_Session_allocateObjectByIdI(
                this, response), id);
    }

    virtual void
    releaseObject(const Identity& id,
        const Current&)
    {
        {
            Lock lock(*this);
            --_allocated;
            assert(_allocated >= 0);
        }
        _session->releaseObject(id);
    }

    void
    allocationFailed()
    {
        Lock lock(*this);
        --_allocated;
        assert(_allocated >= 0);
    }
    // . . .

private:

    const Glacier2::SessionControlPrx _control;
    const IceGrid::SessionPrx _session;
    int _allocated;
};
typedef IceUtil::Handle<SessionI> SessionIPtr;

class AMI_Session_allocateObjectByIdI :
    public AMI_Session_allocateObjectById
{
public:

    AMI_Session_allocateObjectByIdI(
        const SessionIPtr& session,
        const AMD_Session_allocateObjectByIdPtr&
        response) :
    _session(session),
    _response(response)
    { }
```

```
    virtual void
    ice_exception(const Exception& e)
    {
        _session->allocationFailed();
        _response->ice_exception(e);
    }
    // . . .
private:
    const SessionIPtr _session;
    const AMD_Session_allocateObjectByIdPtr
        _response;
};
```

Next, we need to change the client to do multiple encodings in parallel. The strategy is to add a pool of worker threads to handle the encodings concurrently. To do this we'll add three classes:

- an `Encoder` object that encodes a single WAV file,
- an `EncoderManager` that manages the set of remaining encoding objects and encoder worker threads,
- an `EncodingThread` that that de-queues the next available unit of work from the encoder manager and calls on the encoder object to perform the encoding.

The implementation of the `EncodingManager` and `EncodingThread` objects is straightforward; see the source code for details.

The `Encoder` object is defined as follows:

```
// C++
class Encoder : public IceUtil::Shared
{
public:
    Encoder(const IceGrid::SessionPrx& session,
        const string& file);
    void run();

private:
    const IceGrid::SessionPrx _session;
    const string _file;
};
typedef IceUtil::Handle<Encoder> EncoderPtr;
```

The encoder object is given the `IceGrid::Session` object, from which it allocates an encoder factory, creates an encoder, encodes the file, and then subsequently releases the encoder factory object.

```
// C++
void
Encoder::run()
{
       // ... Prepare the WAV file for encoding.
       ObjectPrx obj =
         _session->allocateObjectByType(
              Mp3EncoderFactory::ice_staticId());
       Mp3EncoderFactoryPrx factory =
         Mp3EncoderFactoryPrx::checkedCast(obj);
       Mp3EncoderPrx encoder =
         factory->createEncoder(
              sfinfo.channels, sfinfo.samplerate);
       try
       {
              // ... Encode the MP3 file.
       }
       catch(const Exception&)
       {
              // ... Do whatever logging is
              // appropriate.
       }
       _session->releaseObject(
              obj->ice_getIdentity());
}
```

At startup, the client creates the IceGrid session, then creates the encoder manager, and adds an encoder object for the file to be encoded:

```
// C++
IceGrid::SessionPrx session = ...;
SessionRefreshThreadPtr refresh =
    new SessionRefreshThread(
        IceUtil::Time::seconds(router->
            getSessionTimeout()/2), session);
refresh->start();
EncodingManagerPtr manager =
    new EncodingManager(3);
for(int i = 1; i < argc; ++i)
{
    manager->add(new Encoder(session, argv[i]));
}
manager->waitForComplete();
manager->destroy();

refresh->destroy();
refresh->getThreadControl().join();

session->destroy();
```

Conclusion

The technique of interposing a session implementation allows you to impose additional restrictions on the client without changing any client side source code. Note that the server in this application is not yet fool-proof. For example, it is still possible for a client to allocate an encoder factory and encode multiple files using this encoder, thus getting more than its fair share of server resources. Further, the server is at the mercy misbehaved clients. For example, a client can allocate an encoder and then never send it data, or send the data very slowly. In the next article in this series, I will present a different design that further isolates the server from ill-behaved clients.

Write Once, Read Everywhere

Michi Henning, Chief Scientist

If you have installed an Ice binary distribution and you look in the `doc/reference` directory, you will find a lot of HTML files, including a file `index.html`. This file contains a table of contents for the Ice reference documentation, which provides the reference material for all the Slice definitions used by Ice and its services, as well as a list of all the properties and a reference for stringified proxies and endpoints. If you have not discovered this documentation yet, I suggest you have a look—this information is useful while you are programming and want to confirm some detail of an Ice API—because HTML is extensively hyperlinked, it makes it easy to navigate among different Slice definitions and quickly home in on the part of the API you are interested in.

Not by coincidence, the HTML reference is exactly the same as appendixes B to D in the Ice manual; it's just that the appendixes in the manual are in PDF format instead of HTML. Obviously, here at ZeroC, we know better than to manually maintain two versions of the same documentation in different formats. Instead, we generate the documentation from SGML in DocBook format. In turn, for the Slice reference, the SGML is generated from Slice definitions by a compiler. This compiler, `slice2docbook`, scans for special documentation comments in Slice definitions and produces SGML in DocBook format from these comments. Here is an example:

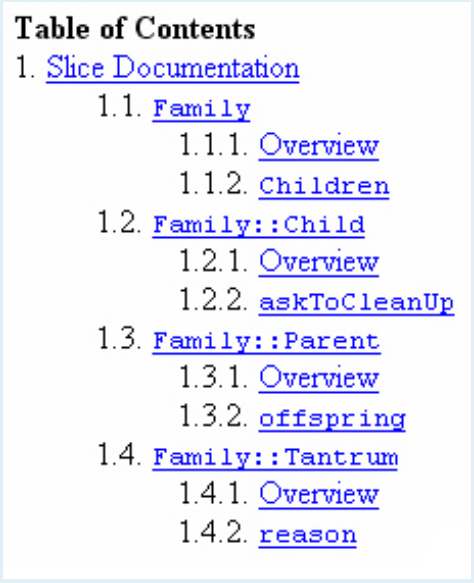
```
// Slice

/**
 * Having a family is like having a bowling
 * alley installed in your head.
 **/
module Family
{
    /**
     * Children frequently throw this.
     **/
    exception Tantrum
    {
        /**
        * Reason for tantrum (often not a
        * discernible one).
        **/
        string reason;
    };

    /**
     * Mother Nature is wonderful. She gives us
     * twelve years to develop a love for our
     * children before turning them into
     * teenagers.
     * <para>
     * Sons are not meant to like their parents
     * That's what greatchildren are for.
     * <para>
     * Children are a great comfort in old
     * age&mdash;and they help you reach it
     * faster, too.
     *
     * @see Parent
     **/
    interface Child
    {
       /**
        * A teenager is always too tired to hold
        * a dishcloth, but never too tired to hold
        * a phone.
        *
        * @throws Tantrum Raised frequently.
        */
        void askToCleanUp() throws Tantrum;
    };

    /**
     * You can learn many things from children.
     * How much patience you have, for instance.
     **/
    sequence<Child> Children;


    /**
     * Parents are the bones on which children
     * cut their teeth.
     *
     * @see Child
     */
    interface Parent
    {
      /**
          * Return the direct descendants of this
          * parent.
          *
          * @param byName If true, the return value
          * is sorted alphabetically; if false, it
          * is sorted by increasing age.
          *
          * @return The direct descendants. (The
          * returned [Children] sequence has at
          * least one element.)
          **/
         Children offspring(bool byName);
    };
};
```

Keeping the documentation of an API together with its code is generally a good thing. That way, as an API evolves over time, there is a better chance that the documentation will be kept up to date as well. Of course, the idea of generating documentation from code is not new—it dates back as far 1983, to Donald Knuth's WEB system. And, obviously, `slice2docbook` was inspired by Javadoc.

If you look through the comments in the preceding example, you will note that the comments contain various kinds of markup (You may have also inferred that I am currently blessed with teenaged offspring…). The text and markup in these comments determine

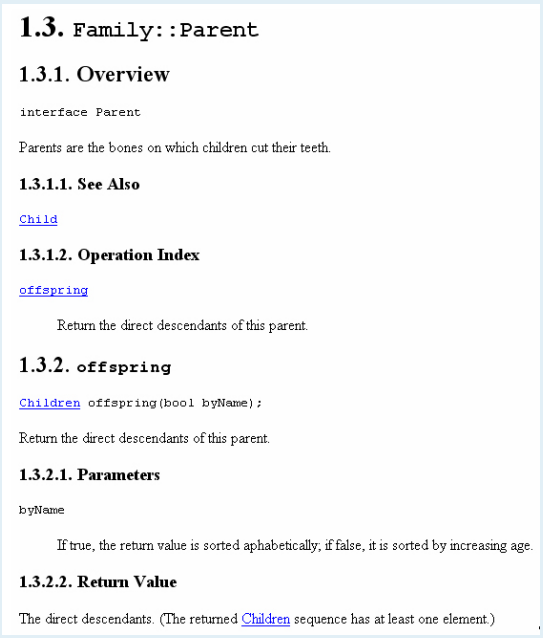
the contents of the generated DocBook output. For example, `slice2docbook` generates the following table of contents for the preceding definitions:

Figure 1: Generated Table Of Contents



Of course, as shown, the text has been post-processed into HTML, so we are not looking at the SGML in DocBook format, but the HTML that we generated from it as displayed by a browser. The underlined parts are hyperlinks to the corresponding section of the documentation. For example, if we follow the `Family::Parent` link, we see the screenshot below.

Figure 2: Family::Parent Section



1.3. `Family::Parent`

1.3.1. Overview

`interface Parent`

Parents are the bones on which children cut their teeth.

1.3.1.1. See Also

Child

1.3.1.2. Operation Index

offspring

Return the direct descendants of this parent.

1.3.2. `offspring`

`Children offspring(bool byName);`

Return the direct descendants of this parent.

1.3.2.1. Parameters

byName

If true, the return value is sorted aphabetically; if false, it is sorted by increasing age.

1.3.2.2. Return Value

The direct descendants. (The returned Children sequence has at least one element.)

Again, this shows the documentation as it appears after being converted to HTML and displayed by a browser. (I will say more on this shortly.)

Markup

Any comment that opens with `/**` and closes with `**/` is a documentation comment. `slice2docbook` only processes text within documentation comments, and ignores normal C-style comments (`/* ... */`) and C++-style comments (`//`). You can add a documentation comment to any Slice construct, such as a module, structure, structure field, and so on. The documentation comment for a Slice construct must precede the definition of that construct.

As for Javadoc, the first sentence of a documentation comment is taken to be a summary sentence. That sentence is used by `slice2docbook` for the generation of a symbol index: the first sentence appears as the summary in the index entry for each symbol.

Hyperlinks

Within a documentation comment, `slice2docbook` recognizes Slice symbols enclosed in square brackets and displays these as a hyperlink. For example, the comment

```
(The returned [Children] sequence has at least one element.)
```

causes the symbol `Children` to be rendered as a hyperlink to the corresponding section of the documentation.

Explicit Cross-References

The `@see` directive must be followed by a single Slice symbol. All `@see` directives for a Slice construct are collected by the compiler and presented in a separate "See Also" sub-section in the documentation for the corresponding construct. You can use several `@see` directives for a single construct. For example,

```
@see Parent
@see Child
```

results in a comma separated list of hyperlinks in the "See Also" section.

Markup for Operations

`slice2docbook` recognizes three directives specifically for operations: `@param`, `@return`, and `@throws`. These directives create separate sub-sections titled "Parameters", "Return Value", and "Exceptions", respectively. For clarity, you should document parameters in the same order as they are defined in the operation signature.

General Markup

You can insert any valid DocBook markup into documentation comments. For example, you can add `<para>` elements to create paragraphs, use `<literal>` elements to display text in constant-width font, and use markup such as `&mdash;` to insert special characters.

Using `slice2docbook`

To create DocBook documentation from Slice files, you specify an output file and an input file. For example:

```
slice2docbook family.sgml family.ice
```

This compiles the documentation comments in `family.ice` and places the DocBook output into `family.sgml`. If you have several Slice files with documentation comments that refer to symbols in other source files, you must pass all of the Slice files on the command line in a single invocation of `slice2docbook`; this is necessary so the compiler can correctly generate hyperlinks that span files. (Please refer to the Ice manual for information on command-line options.)

Generating HTML

Once we have generated SGML in DocBook format with `slice2docbook`, we can render the documentation into various formats, such as HTML, PDF, and others. To do this, you can grab any one of a number of format converters. For Ice, we use db2html, which is available with most Linux distributions. (If you prefer to use other tools, you can find many of them on the web.)

To get an idea of how to generate HTML from Slice, have a look at the example code for this article. The code contains a `Makefile` that invokes `slice2docbook` and `db2html`. Instead of compiling the generated `family.sgml` file directly, the example compiles a file called `reference.sgml` that specifies the DTD for Docbook, and then includes `family.sgml` as a `<chapter>` within a `<book>` element:

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook
V3.1//EN" [
  <!ENTITY SliceDoc SYSTEM "family.sgml">
]>

<book><?dbhtml filename="index.html">
  <chapter>
    <title>The Model Family</title>
    &SliceDoc;
  </chapter>
</book>
```

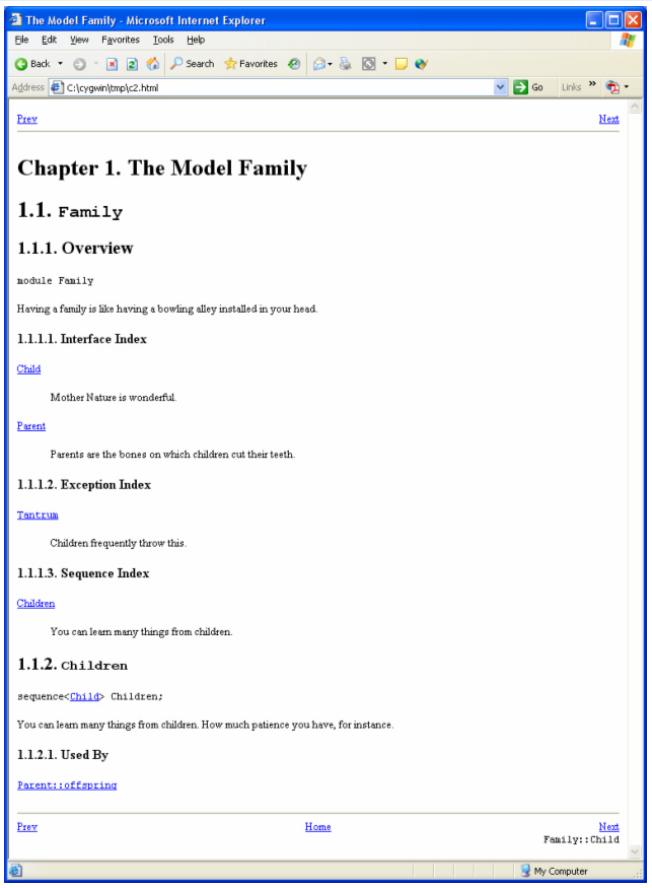
By doing this, we end up with a well-formed document in DocBook format because `slice2docbook` uses `<section>` as the top-level element in the generated SGML. The actions for `make` boil down to the following two commands:

```
slice2docbook family.sgml family.ice
db2html reference.sgml
```

`db2html` places the generated HTML into a `reference` subdirectory; in this directory, you will find `index.html`, plus a separate HTML file for each Slice construct.

The generated HTML makes it easy to peruse the documentation. For example, here is a screenshot of the `Family` module overview:

Figure 3: Family Modufle Overview



Generating Other Formats

DocBook documentation can easily be converted to a large number of other popular formats. For example, generating PDF is as simple as replacing `db2html` with `db2pdf`. If you have FrameMaker, you can import the generated SGML into a FrameMaker document and then use FrameMaker's built-in tools to control layout and look of the text. This is what we do for the PDF version of the Ice manual: the manual is authored in FrameMaker, and appendixes B to D are created by simply importing the generated SGML. The appearance of the appendixes is controlled by an element definition document that maps DocBook elements to FrameMaker paragraph and character tags. (See the FrameMaker documentation for details on how to do this.) There are many other documentation formats that you can create from DocBook, among them PostScript, RTF, TeX, DVI, and man pages. Tools that can target these and other formats are freely available.

Summary

`slice2docbook` allows you to keep your Slice definitions and documentation in one place, making it less likely for discrepancies between the API and its documentation to creep in. You can easily convert the SGML that is generated by `slice2docbook` into a large number of other formats. In turn, this makes it easy to integrate Slice documentation with the remainder of your documentation system, and it provides you with flexibility if you want to publish documentation in new formats as your application evolves over time. If you want to publish documentation for your Slice definitions, I suggest you give `slice2docbook` a try—it truly allows you to "write once, and read everywhere".

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: The Slice compiler does not work—what should I check?

The various Slice compilers (`slice2cpp`, slice2java, and so on) launch the C preprocessor `icecpp.exe` as a sub-process. On Windows, if the directory containing `icecpp.exe` is not in your `PATH`, you can get an error message:

```
> slice2cpp Hello.ice
'icecpp.exe' is not recognized as an internal or
external command,
operable program or batch file.
```

The same problem can arise under UNIX:

```
$ slice2cpp Hello.ice
sh: line 1: icecpp: command not found
```

Another common problem on Windows is an incorrect setting of the `ComSpec` environment variable. This variable specifies the location of the command line interpreter and, if incorrectly set, prevents the Slice compiler from being started by the development environment. For example, with Visual Studio, if `ComSpec` is set incorrectly, the build halts with an error:

```
------ Build started: Project: test,
Configuration: Debug Win32 ------

Performing Custom Build Step
Project : error PRJ0019: A tool returned an error
code: "Performing Custom Build Step"
```

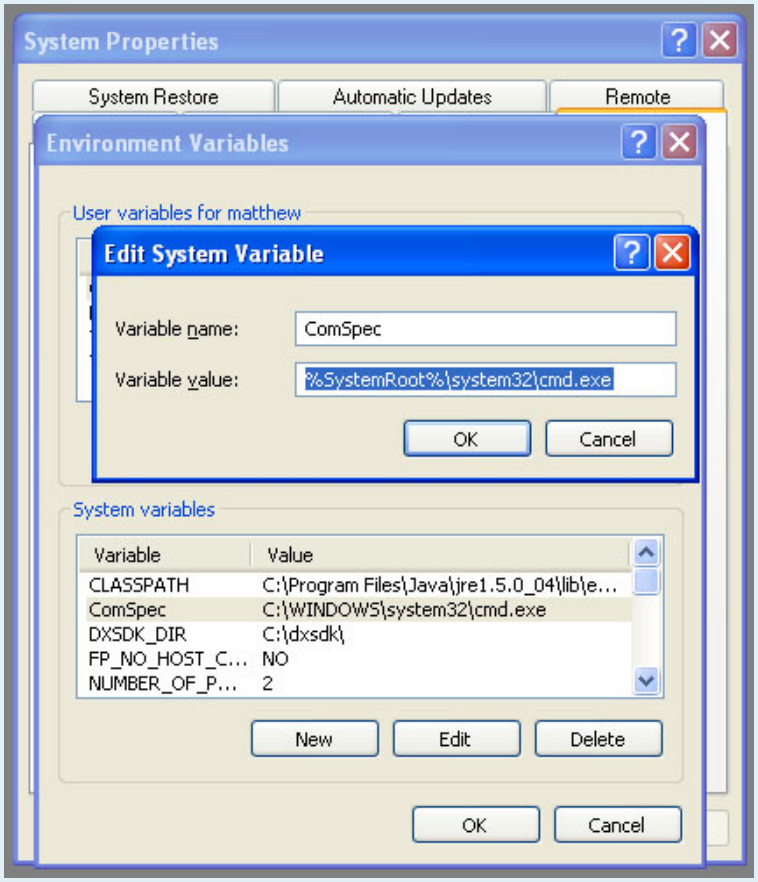
You can check the value of the `ComSpec` environment variable by opening a command window and displaying its value:

```
> echo %ComSpec%
C:\WINDOWS\system32\cmd.exe
```

Unless you are using a custom command line interpreter or have installed Windows in a location other than the default, the value should be the one shown above.

If `ComSpec` is incorrectly set and you need to change it, you can right-click on "My Computer" in the start menu and select "Properties" to open the system properties window. Select the "Advanced" tab and then click on "Environment Variables". In the "System Variables" section, select the `ComSpec` variable and press Edit. It should look similar to the following:

Figure 1: Edit System Variable



You should change the value to:

```
%SystemRoot%\system32\cmd.exe
```

Press OK, and then OK again in the "Environment Variables" dialog. Note that, for the setting to take effect, you must restart any process that depends on the new value. For example, if you use Visual Studio, you need to close it down and restart it for it to recognize the new variable setting. Similarly, if you are using a command prompt and have changed the PATH setting, you need to start a new command window for the changed setting to take effect.

Q: How do I use `Ice.loadSlice()` with Ice for Python?

Ice for Python supports two types of code generation: static and dynamic. Static code generation, which will be familiar to users of compiled languages such as C++ and Java, uses the Slice compiler `slice2py` to create Python source files from Slice definitions at compile time. On the other hand, when a program calls the `Ice.loadSlice` function, code is generated from Slice definitions dynamically at run time. Dynamic code generation slightly increases program start-up time but eliminates the need to generate and manage additional Python source files. (Please see the Ice manual for a discussion of the trade-offs of static and dynamic code generation.)

Users of dynamic code generation sometimes encounter problems when multiple Slice files are involved. For example, consider the following Slice definitions:

```
// A.ice
module A
{
    struct Pair
    {
       long first;
       long second;
    };
};

// B.ice
#include <A.ice>
module B
{
    interface Mapper
    {
      void doit(A::Pair p);
    };
};
```

Now run python as follows:

```
$ python
Python 2.3.5 (#1, Jan 13 2006, 20:13:11)
[GCC 4.0.1 (Apple Computer, Inc. build 5250)] on
darwin
Type "help", "copyright", "credits" or "license"
for more information.
>>> import Ice
>>> Ice.loadSlice('B.ice')
B.ice:1: No include path in which to find A.ice
B.ice:6: `A::Pair' is not defined
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: Slice parsing failed for `B.ice'
```

The problem in this case is that `B.ice` includes `A.ice` but the compiler cannot find `A.ice` because the include path is not set. To remedy this, use the `-I` option to tell the code generator where to find the included files. Assuming `A.ice` resides in the same directory as `B.ice`, you can use the following command:

```
>>> Ice.loadSlice('-I. B.ice')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "B.ice", line 3, in ?
    {
ImportError: No module named A_ice
```

Now `A.ice` is found, but `loadSlice` complains about an import error. As explained in the Ice manual, the reason for this error is that, by default, the code generator creates Python code only for the Slice definitions in the specified file; for included files, it instead generates equivalent Python `import` statements. In other words, by default, the compiler assumes that included files have already been translated statically. (The reason for this behavior is that it allows you to combine dynamic and static code generation when necessary.)

To get the compiler to generate code for both `B.ice` and the included `A.ice` file, we need to specify the `--all` option:

```
>>> Ice.loadSlice('-I. --all B.ice')
```

This recursively translates `B.ice` and all files it includes.