



Leaky Abstractions

When I wrote last [month's editorial](#), I was going to add a section on “leaky” abstractions but, after running out of space, decided to make that the topic of this month's editorial instead. To my surprise, after we published [Issue 18](#), Dilip Ranganathan got in touch with me and asked whether I knew of [Joel Spolky's law of leaky abstractions](#)

(which I had never heard of before). It seems that Joel independently came up with the same idea, namely that abstractions are “leaky”. As it turns out, I gave presentations that mentioned leaky abstractions in [June 2001](#) and [November 2001](#), so I managed to beat Joel to the punch by a few months. (Incidentally, the latter presentation also marks my departure from CORBA—it is the keynote I gave at the OMG meeting in Dublin, which was the last meeting I attended.)

So, what's the big deal about leaky abstractions? They leak their secrets, that is to say, they are not perfect. Both Joel and I quoted virtual memory as an example of a leaky abstraction: virtual memory provides the illusion of memory very much larger than physical memory. Most of the time, I can afford to forget that virtual memory is not real memory, and write my code as if they were both the same thing. However, sometimes, I cannot: Years ago, I had a new graduate complain to me that “The code works, except it's terribly slow.” When I looked at his code, I found that he had used `mmap()` to implement a very large sparse array and, as a result, the machine was page faulting itself into oblivion.

Here is my favorite example of a leaky abstraction:

```
char c;
short i;
Mutex cMutex; // Protects c
Mutex iMutex; // Protects i
// ...
cMutex.lock(); // Thread 1
c = getchar();
cMutex.unlock();
// ...
iMutex.lock(); // Thread 2
++i;
iMutex.unlock();
```

Even though the two threads that update `c` and `i` faithfully lock the appropriate mutex, every now and then, the state of the two variables gets corrupted. (If you have not come across this before, take a moment to see whether you can work out why...)

The corruption occurs if `c` and `i` happen to occupy the same word of memory: the underlying hardware is simply incapable of updating `c` without also updating `i` and, depending on the timing of the two threads, they will sometimes write to the same memory cell concurrently, despite the locks. (This particular problem is known as a [word-tearing race](#).)

All abstractions leak to some extent, and Ice is no exception. For example, even though an Ice RPC looks just like a local procedure call, you cannot afford to forget that it is not. An RPC is around four orders of magnitude slower than a local call, and it has different semantics: while a local call can fail only if the program's state is corrupted or the hardware is faulty (in which case the program is toast anyway), an RPC can fail for all sorts of external reasons.

What this means is that you cannot design an Ice application as you would a non-distributed one. The leaks in the abstractions matter and you must create your design with this in mind. But, provided that you do, you can enjoy the abstractions for all they are worth. Or, to paraphrase a common idiom: an abstraction in the hand is worth two leaks in the roof...

Michi Henning
Chief Scientist

Issue Features

Custom Sessions and IceGrid

In this article we show how to make your server application more resistant to client side abuse through the use of a custom session

Teach Yourself IceGrid in 10 Minutes

Michi Henning describes the basics of IceGrid and why your applications should use it.

Contents

Custom Sessions and IceGrid	2
Teach Yourself IceGrid in 10 Minutes	11
FAQ Corner	16

Custom Sessions and IceGrid

Matthew Newhook, Senior Software Engineer

Introduction

In the previous article, we added an interposed session to our encoder application so clients could use only a fixed number of MP3 encoders, to prevent them from using more than a fair share of resources. However, the server still has a serious flaw: If the encoding process only occurs over a LAN, there will typically be no problem because each client has a very fast connection to the server backend. However, over a WAN, with limited bandwidth, the situation is different: a client with a slow connection can occupy an encoder much longer than a client with a fast connection. This is unacceptable because the speed of the connection to the client should not play any part in the allocation time of a back-end server that is otherwise primarily CPU bound.

A malicious client could do even more damage by allocating an MP3 encoder but never using it. As long as the client keeps the session alive, it can tie up resources indefinitely. Of course, this could be rectified in a number of ways. For example, we could try to ensure that the MP3 data is streamed at a minimum guaranteed rate, or bill for allocated time as well as bytes encoded, among other things. However, these solutions ignore the key problem, namely, that server side encoding resources are occupied while the WAV data is streamed to and from the server.

In order to solve this problem, we are going to split the data transfer and the encoding of the MP3 byte stream into separate steps. First we send all of the data to the server, and then we encode it.

Let's redesign the client interfaces to meet this goal. First, we want to ensure that all of the MP3 data is present on the server side before the process of encoding the data starts. This suggests an interface such as the following:

```
// Slice
interface Encoder
{
    Ice::ByteSeq encode(Samples left,
                       Samples right);
};
```

This interface suffers from the problem that it sends all of the samples and receives all of the encoded data in a single invocation. Because the amount of data is considerable, this is unlikely to work. As explained in the FAQ [“How do I transfer a file with Ice?”](#), it is better to send the data in chunks. The FAQ recommends the following interface:

```
// Slice
interface FileStore
{
    ByteSeq read(string name, int offset,
                int num);
    void write(string name, int offset,
              ByteSeq bytes);
};
```

This is appropriate for a file store but is not all that convenient for sending a WAV file to the server and receiving the corresponding MP3-encoded byte sequence. Here is a modified version of the interface:

```
// Slice
interface Encoder
{
    void encode(Samples leftSamples,
              Samples rightSamples);
    void destroy();
};
```

The client repeatedly calls `encode`, passing samples in chunks. Once it has sent all of the samples, it calls `destroy` to indicate that the samples are complete and that encoding should start. To get the results, the client can use a similar interface:

```
// Slice
interface EncodingResult
{
    void result(Ice::ByteSeq bytes);
    void destroy();
};
```

The client passes a proxy for this interface to the server, and the server invokes the `result` operation repeatedly to pass the encoded data to the client, and calls `destroy` once it has sent all of the data.

How does the client interact with the server to create an `Encoder` object? In the previous article, we used an interposed IceGrid session for the client-side interactions to avoid making changes to the client interface. However, with the new interfaces, we have a very different interaction model, so we'll create a custom session interface as follows:

```
// Slice
interface EncodingSession extends Glacier2::
Session
{
    void keepAlive();
    Encoder* create(string desc, int channels,
                  int sampleRate, EncodingResult* result);
};
```

As with the IceGrid session approach, the client must call `keepAlive` on a timely basis to keep the session alive. The client calls `create` to encode a new MP3 file, passing a description of the file to be encoded, the number of channels to encode, the sample rate, and a proxy to the `Result` object; the operation returns a proxy to an `Encoder` object.

Server Architecture

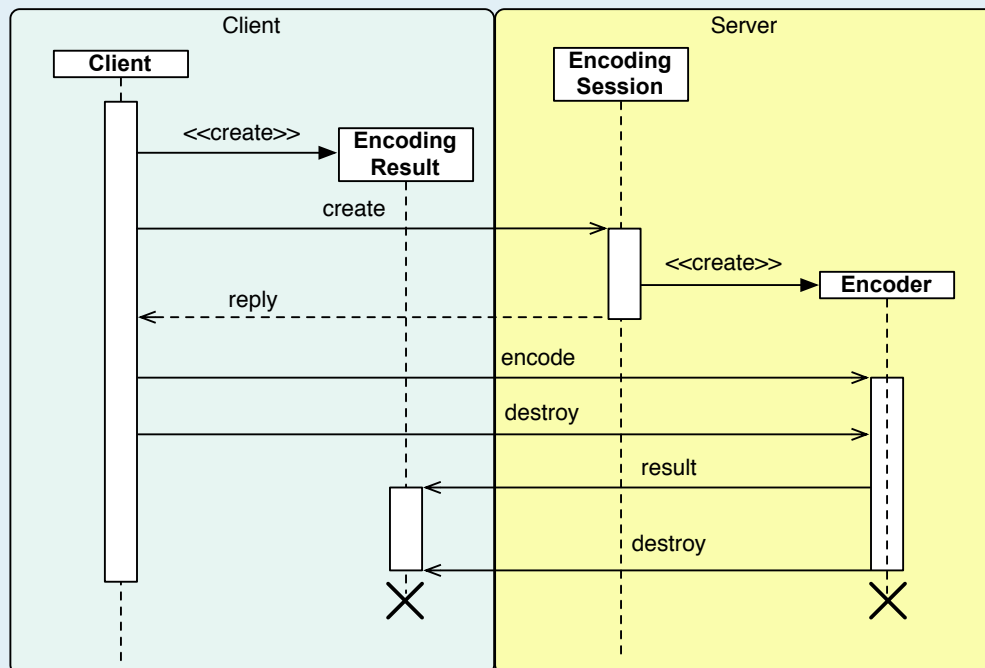
Let's take a look at the server-side implementation. To decide how to proceed, we need to know what objects are necessary for the implementation and consider the client interaction in more detail. See Figure 1 and Figure 2 for object and interaction diagrams of the encoding process.

A `SessionManager` manages multiple `Sessions`. In turn, a `Session` has multiple `Encoder` objects that each have a single `EncodingResult` that is used to return the MP3-encoded data to the client. The client creates an `Encoder` object by calling `EncodingSession::create` and passing a proxy for the result object.

The encoding process proceeds only once the client calls `destroy` on the encoder object. At that point, all of the data is available to the server side and the server can now allocate an MP3 encoder, encode the data, and send the results back to the client. Note that, for the actual encoding process, we can use the same server back end as in the previous articles in this series.

To do the encoding, we'll create a second object called an `EncoderWorkItem` that encapsulates the encoding process. As raw data arrives, the encoder stores the data. Once the client destroys the encoder, the work item is created and uses the `MP3Encoder` to encode the data and then sends the encoded MP3 data back to the client via the associated `EncodingResult` object.

Figure 1: Interaction Diagram



(or at least in the same thread that calls the `destroy` method). That is, once the client calls `destroy`, we allocate the MP3 encoder, encode the transmitted data, and send the results back. However, this has the serious disadvantage of consuming a server-side dispatch thread for an extended period of time. This is not acceptable because, while the server side dispatch thread is busy encoding data, other clients may not be able to invoke on their sessions. Therefore, we instead need a separate thread to do the encoding process. We have two options here:

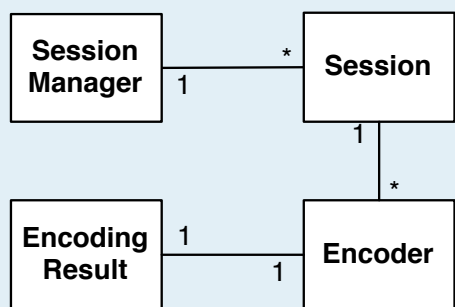
- Create a new thread that manages the encoding process once `destroy` is called.
- Use a pool of worker threads to manage the encoding process.

Note that creating new threads is expensive and, due to stack size limitations on most operating systems, the number of threads that can be created is limited. (See the FAQ [“How can I increase the maximum number of threads my C++ application can create?”](#) for more information on this topic). Consider 100 concurrent sessions, each encoding 20 files. We would need 2000 threads to do this, which is definitely too many. Clearly, if we want scalability, we need to use a thread pool. The question is how to allocate this thread pool.

Once again we have two options:

- Per-session thread pool
- Per-session manager thread pool (shared between each session)

Figure 2: Object Diagram



How should we manage the encoding process? One option would be to do the encoding in the `Encoder::destroy` method

If we allocate the thread pool on a per-session basis, the session itself will be limited to concurrently encoding as many WAV files as there are threads in the thread pool, regardless of how many encoders are available on the server back end. Ideally, we want the back-end encoders do be fully utilized if possible, regardless of the number of clients.

This points to a per-session manager thread pool as the better solution. However, with a shared thread pool, the queuing strategy becomes important: we don't want a single client to queue up a whole host of MP3 files and hog all of the encoders while other clients are waiting for their data to be encoded. Therefore, the fairest solution would probably be a queue per session that is serviced in a round-robin fashion. However, implementing a round-robin per-session queue is fairly complex, so I will not present this solution here; instead, we will use a simple queue of outstanding working items.

The encoding process is managed by a pool of encoding threads that are managed by a per-session manager encoding queue. Once a client destroys an encoder, the encoder places the work item on the encoding queue, which processes the item as soon as a worker thread becomes available—see Figures 3 and 4.

Figure 3: Encoding Object Model

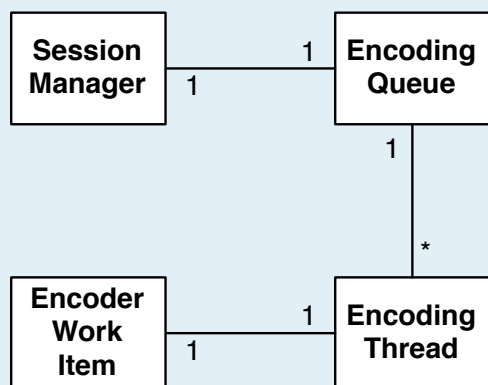
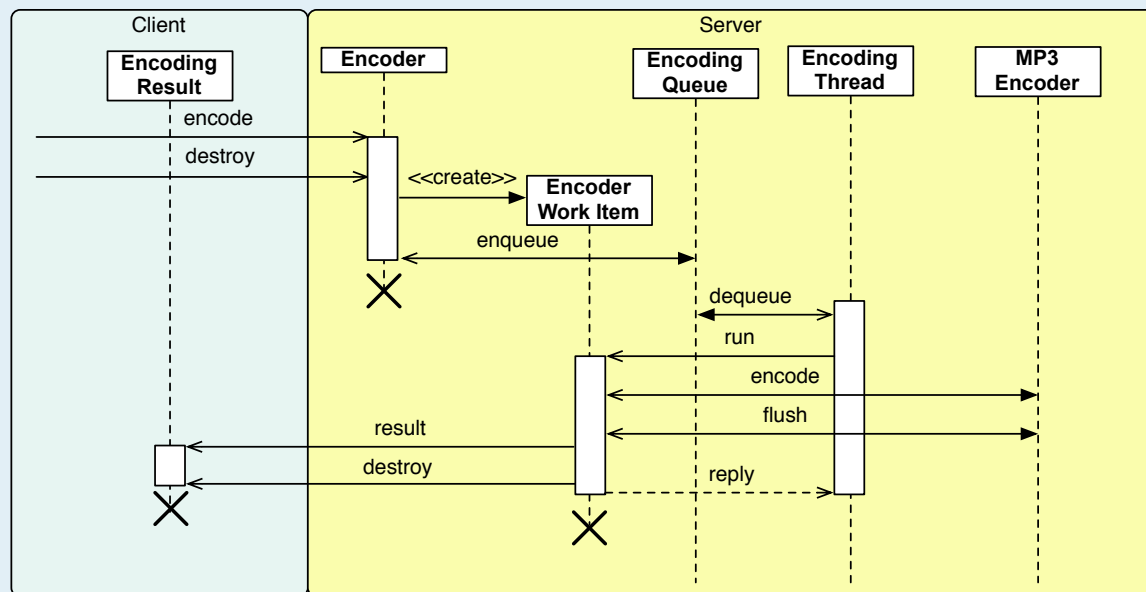


Figure 4: Encoding Interaction Diagram



Consider the process of encoding the data. This will look something like the client-side code I presented earlier:

```

// C++
ObjectPrx obj = _session->allocateObjectByType(
    Mp3EncoderFactory::ice_staticId());
Mp3EncoderFactoryPrx factory =
Mp3EncoderFactoryPrx::checkedCast(obj);

Mp3EncoderPrx encoder =
    factory->createEncoder(channels, samplerate);
while(/*data left to encode*/)
{
    Samples lbuf;
    Samples rbuf;
    // Fill lbuf and rbuf from the received WAV
    // data.
    Ice::ByteSeq bytes = encoder->encode(
        lbuf, rbuf);
    // Send bytes to the client.
}
// ...

```

What strategy should we use in the server to send the encoded bytes back to the client? Consider a straight synchronous method invocation as follows:

```

// C++
EncoderResultsPrx result = ...;
// ...
Ice::ByteSeq bytes = encoder->encode(lbuf, rbuf);
result->result(bytes);

```

This blocks a server thread until the byte sequence has been transmitted to the client, which is bad because transmission can take

quite some time. Moreover, while the data is being transmitted, the back-end encoder sits idle, which is undesirable. Instead, we could use asynchronous invocations:

```
// C++
EncoderResultsPrx result = ...;
// ...
Ice::ByteSeq bytes = encoder->encode(lbuf, rbuf);
result->result_async(
    new AMI_EncoderResult_resultI, bytes);
```

However, there is also problem with this approach. We must not send a second result chunk until after a reply has been received for the preceding request: if we send without waiting for a reply, the client can receive the invocations out of order. (Note that this AMI call cannot block since Glacier2 buffers it).

To get around this, we could buffer up all of the replies and send them once the encoding has completed (and the backend encoder object has been released, thus making it available to any other pending encoders). However, this will consume the worker thread for the duration of the transmission of the entire result back to the client. During that time, if all workers are fully utilized, no other encodings can take place within the session manager. Even worse, a single slow client could end up consuming all worker threads, thus monopolizing an entire front-end session manager.

Buffering up data also has the additional side effect of delaying the transmission of the data back to the client until the entire dataset is encoded, which is inefficient both in terms of bandwidth (assuming that the bandwidth is not being used for some other purpose), and in terms of storage because we'll have to store data that could otherwise be thrown out as soon as it has been transmitted to the client.

Instead, we'll queue the encoded data in a per-client object that sends the data to the client. Why do we use a per-client object, and not a per-encoder result object? The reason is that there is no benefit in allowing multiple threads to send the same client at the same time. (In fact, it would likely slow things down due to increased context switching.)

Next, we need to decide how to send the messages. The obvious approach is to use a sender thread that removes messages from the pending queue and sends them one at a time. While this approach would certainly work, there is another approach that I want to explore.

Instead of using a separate thread, we will send each message asynchronously. We use the `ice_response` call on the AMI callback object to trigger the sending of the next message in the response queue. That way, we avoid having to use a separate thread. With this scheme, we send a message under two circumstances:

- When a new message is pushed onto the queue and no response is pending.
- When a response is received and there are pending messages on the queue.

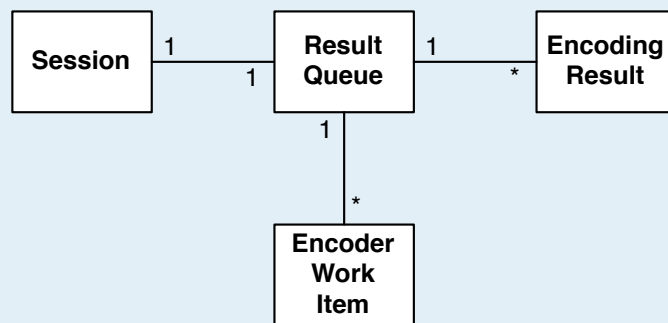
By using the thread that calls the AMI callback (which comes from the Ice client side thread pool), we avoid the overhead of spawning an additional thread ourselves. You may wonder whether AMI presents a problem because AMI calls can potentially block? The answer is no: because calls to Glacier2 can be considered safe and should never block (unless we have some serious internal network problem, in which case probably nothing works anyway), we can safely use AMI.

We'll look at the exact implementation of this sender object shortly. The encoder worker looks something like this:

```
// C++
while(/*data left to encode*/)
{
    Samples lbuf;
    Samples rbuf;
    // Fill lbuf and rbuf from the received WAV
    // data.
    Ice::ByteSeq bytes = encoder->encode(
        lbuf, rbuf);
    _sender->queue(bytes);
}
// ...
```

Figure 5 shows the object diagram.

Figure 5: Result Queue Object Diagram



Server Implementation

For brevity, the description that follows does not show all of the implementation. Instead, I have concentrated on the important highlights.

Firstly, we'll look at the implementation of the session manager. The session manager must implement the `Glacier2::SessionManager` to create the encoding session:

```
// Slice
module Glacier2
{
    interface SessionManager
    {
        Session* create(string userId,
```

```

    SessionControl* control)
    throws CannotCreateSessionException;
};
};

```

As discussed in my article in [Issue 18](#) of *Connections*, Glacier2 validates the user before it creates a session, and you can use the session control object to control Glacier2 filters. (I recommend reviewing this article before continuing.) At some point, the session will need to allocate IceGrid objects for the actual encoding, so we first need to create an IceGrid session. This suggests a class definition as follows:

```

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

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

private:
    const Glacier2::SessionManagerPrx _manager;
    std::vector<Glacier2::SessionPrx> _sessions;
    EncodingQueuePtr _encodingQueue;
};

```

The `_manager` data member contains a proxy to the IceGrid session manager. The session also keeps track of what sessions have been created, and maintains the encoding queue in the `_encodingQueue` data member. We also have a `destroy` method that is called when the session manager shuts down. We need the `destroy` method because, otherwise, we could not correctly reclaim resources (such as IceGrid sessions) in response to an orderly shutdown. (In the event of a crash, the IceGrid session is reclaimed because it will time out).

Now we can move onto the implementation of the `create` method:

```

// C++
Glacier2::SessionPrx
SessionManagerI::create(const string& userId,
    const Glacier2::SessionControlPrx& control, const
    Ice::Current& current)
{
    Lock sync(*this);

    //
    // Reap any dead sessions.
    //
    vector<Glacier2::SessionPrx>::iterator p =
        _sessions.begin();
    while(p != _sessions.end())

```

```

{
    try
    {
        (*p)->ice_ping();
        ++p;
    }
    catch(const Ice::Exception&)
    {
        p = _sessions.erase(p);
    }
}

```

First, we run through all created sessions and reap any that have been destroyed. We do this so that the session itself does not need to call back on the session manager object. Arranging the call graph of objects to create an acyclic graph (that is, to avoid call-backs) is a commonly used method for avoiding deadlocks—see Bernard’s articles in [Issue 4](#) and [Issue 5](#) of *Connections* for more details.

Here is the `create` operation for the session manager:

```

// C++
Glacier2::SessionPrx
SessionManagerI::create(
    const string& userId,
    const Glacier2::SessionControlPrx& control,
    const Ice::Current& current)
{
    // . . .
    IceGrid::SessionPrx gridSession =
        IceGrid::SessionPrx::uncheckedCast(
            _manager->create(userId, control));
    Ice::LoggerPtr logger =
        current.adapter->
            getCommunicator()->getLogger();
    Glacier2::SessionPrx session =
        Glacier2::SessionPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new SessionI(
                    logger, control, gridSession,
                    _encodingQueue)));
    _sessions.push_back(session);

    Glacier2::IdentitySetPrx identities =
        control->identities();
    Ice::IdentitySeq ids;
    ids.push_back(session->ice_getIdentity());
    identities->add(ids);
    ids.clear();
    ids.push_back(gridSession->ice_getIdentity());
    identities->remove(ids);

    return session;
}

```

`create` allocates an IceGrid session as well as our custom session object. It then alters the Glacier2 filtering rules to add the newly created session to the set of permitted objects, and it removes the IceGrid session from the set of permitted objects. (See my article [“Session Management with IceGrid”](#) for more details on the neces-

sity of altering the Glacier2 filtering rules.)

Next we look at the session implementation. First, the class definition:

```
// C++
class SessionI : public EncodingSession,
    public IceUtil::Mutex
{
public:
    SessionI(
        const Ice::LoggerPtr&,
        const Glacier2::SessionControlPrx&,
        const IceGrid::SessionPrx&,
        const EncodingQueuePtr&);
    ~SessionI();

    virtual void ice_ping(
        const Ice::Current& current);
    virtual void keepAlive(const Ice::Current&);

    virtual EncoderPrx create(
        const string&, int, int,
        const EncodingResultPrx&,
        const Ice::Current&);
    virtual void destroy(const Ice::Current&);

private:
    const Ice::LoggerPtr _logger;
    const Glacier2::SessionControlPrx _control;
    const IceGrid::SessionPrx _session;
    const EncodingQueuePtr _manager;
    const EncodingResultQueuePtr _queue;
    vector<pair<Ice::Identity,
        EncoderWorkItemPtr> > _encoders;
};
```

Here is the implementation of keepAlive and ice_ping:

```
// C++
void
SessionI::keepAlive(const Ice::Current& current)
{
    try
    {
        _session->keepAlive();
    }
    catch(const Ice::ObjectNotExistException&)
    {
        destroy(current);
        throw;
    }
}

void
SessionI::ice_ping(const Ice::Current& current)
{
    try
    {
        _session->ice_ping();
    }
}
```

```
    }
    catch(const Ice::ObjectNotExistException&)
    {
        destroy(current);
        throw;
    }
}
```

keepAlive is straightforward. When the client side calls keepAlive on the session, we in turn call keepAlive on the IceGrid session. If the call fails, the IceGrid session is dead, so we in turn destroy our own session and inform the client.

ice_ping is more interesting. The implementation is the same as keepAlive, except that it calls ice_ping on the IceGrid session. But why do we bother with this call? If you recall the reaping of sessions in the session manager, the manager runs through all sessions and calls ice_ping on each session to determine if the session is still alive. Now consider the situation of a Glacier2 crash. In that case, all clients are kicked off, and the sessions will eventually time out because keepAlive is no longer called. However, note that keepAlive does not record a timeout, so how does this work? Our implementation relies on IceGrid to time out the session. By delegating the ice_ping call to the IceGrid session, we can detect when the IceGrid session disappears and subsequently destroy our own session. In case IceGrid itself becomes unreachable, we will not destroy the session until IceGrid comes back up, but this is not a concern because the whole service is useless without IceGrid anyway. The session create operation looks as follows:

```
// C++
EncoderPrx
SessionI::create(
    const string& desc,
    int channels, int samplerate,
    const EncodingResultPrx& result,
    const Ice::Current& current)
{
    Lock sync(*this);

    EncoderPrx encoder =
        EncoderPrx::uncheckedCast(
            current.adapter->addWithUUID(
                new EncoderI(_manager, _logger, _session,
                    result, _queue, desc, channels,
                    samplerate)));
    _encoders.push_back(
        encoder->ice_getIdentity());
    Ice::IdentitySeq ids;
    ids.push_back(encoder->ice_getIdentity());
    _control->identities()->add(ids);

    return encoder;
}
```

create creates a new work item and an encoder servant, and it adjusts the Glacier2 filtering rules to allow access to the new object.

Here is the implementation of `destroy`:

```
// C++
void
SessionI::destroy(const Ice::Current& current)
{
    Lock sync(*this);
    try
    {
        current.adapter->remove(current.id);
    }
    catch(const Ice::NotRegisteredException&)
    {
        return;
    }
    _queue->destroy();
    Ice::IdentitySeq ids;
    ids.push_back(current.id);
    vector<Ice::Identity>::const_iterator p;
    for(p = _encoders.begin();
        p != _encoders.end();
        ++p)
    {
        try
        {
            ids.push_back(*p);
            current.adapter->remove(*);
        }
        catch(const Ice::NotRegisteredException&)
        {
            // Ignore. If the encoder is already
            // destroyed this can be expected.
        }
    }
    _encoders.clear();
    try
    {
        _control->identities()->remove(ids);
    }
    catch(const Ice::Exception&)
    {
    }
    try
    {
        _session->destroy();
    }
    catch(const Ice::Exception&)
    {
    }
}
```

`destroy` first removes the servant from the object adapter. It then destroys the result queue, which prevents any further encoded results from being forwarded to the client. `destroy` then runs through the created encoders and removes them from the object adapter. Finally, the code removes all the registered objects from the Glacier2 filters and destroys the IceGrid session.

The implementation of the encoder is straightforward. The `encode` method adds the left and right channel samples to an

internal buffer. (See the discussion below regarding memory and secondary storage.) The `destroy` method removes the servant from the object adapter, and then creates and queues a work item with the encoding queue. (See the source code for details.)

The implementation of the encoder work item is very similar to what I discussed previously. However, there are a few interesting things worth mentioning.

Firstly, this implementation buffers all of the data in memory in two vectors of samples. If your front end has loads of memory, this is appropriate. However, more likely, you would store the samples in secondary storage. This is not very difficult (though you must ensure that you reclaim this secondary storage correctly in the event of a crash). Secondly, because we have the data available in a vector, we can make use of the alternative C++ array mapping supported by Ice in order to avoid making an extra copy of the data for transmission:

```
// C++
interface Mp3Encoder
{
    // Input: PCM samples for left and right
    // channels Output: MP3 frame(s).
    Ice::ByteSeq encode(
        ["cpp:array"] Samples leftSamples,
        ["cpp:array"] Samples rightSamples)
        throws EncodingFailedException;
    // . . .
};
```

Now, when calling the `encode` method, we provide a pair of `Ice::Short` pointers, the first of which points to the start of the buffer, and the second of which points one element past the end of the buffer (just like an STL iterator). Thus the primary encoding loop is as follows:

```
// C++
// Contains the samples.
Ripper::Samples _left, _right;
// The encoder result queue.
EncodingResultQueuePtr _queue;
// The encoding result proxy.
EncodingResultPrx _result;

MP3EncoderPrx encoder = ...;
Ripper::Samples::size_type curr = 0;
int nsamples = (1000 * 1000) / 8;
while(curr < _left.size())
{
    if(_queue->destroyed())
    {
        throw EncoderDestroyedException(
            __FILE__, __LINE__);
    }
    int max = nsamples;
    if(curr + nsamples > _left.size())
    {
        max = _left.size() - curr;
    }
}
```

```

encoded = encoder->encode(
    make_pair(&_left[curr], &_left[curr+max]),
    make_pair(&_right[curr],
        &_right[curr+max]));
curr += max;
_queue->result(_result, encoded);
}

```

Note the call to `_queue->destroyed()` as each set of samples is encoded. In the event that the hosting session is destroyed, this marks the encoding result queue destroyed as well. We check this flag in each iteration and terminate the encoding process if the queue has in fact been destroyed.

Before we can look at the encoding result queue object, we have to deal with error handling. In case of an error, thus far, the client had no way to tell the server that it cannot continue to deal with the encoding results. Consider an implementation of the `EncoderResult` object:

```

// C++
class EncodingResultI : public EncodingResult
{
public:
    EncodingResultI(FILE* fp) :
        _fp(fp)
    {
    }

    virtual void
    result(const Ice::ByteSeq& bytes,
        const Ice::Current&)
    {
        if(fwrite(&bytes[0], 1, bytes.size(), _fp)
            != bytes.size())
        {
            // What to do here?
        }
    }
    // ...
private:
    FILE* _fp;
};

```

What can `result` do if it the write to a file fails? Most likely, the failure is due to a file system error, such as running out of disk space, and all future writes will also fail. In that case, there is little point in continuing with the encoding process. One option would be to terminate the session and abort the client. However, it is nicer to inform the server of the problem with an exception, as follows:

```

// Slice
exception ResultException
{
    string reason;
};

interface EncodingResult
{
    void result(Ice::ByteSeq bytes)

```

```

        throws ResultException;
    void destroy();
    // . . .
};

```

In addition, it would be nice for the server to have an operation on the encoding result that notifies the client in the event of a failure. For example, if server encounters an error such as failure to allocate an MP3 encoder, it should let the client know about it. We cannot easily do this with an exception because exceptions cannot be passed as parameters to a callback operation. Instead, we'll add a `failed` method as follows:

```

// Slice
interface EncodingResult
{
    // ...
    void failed(string reason);
};

```

If anything goes wrong, the server calls `failed` and provides a description of the error in the `reason` parameter. (The client should destroy the encoding result object in response to a call to `failed`.)

Now we can proceed with the implementation of the encoding result queue. This object holds a queue of pending messages to be sent to a client. A message consists of an encoding result proxy and an MP3-encoded byte sequence, or a call to `destroy`.

The following is a simplified version of the `EncodingResultQueue`. I have glossed over some of the more complex issues, such as error handling; see the [accompanying source code](#) for full details.

```

// C++
class EncodingResultQueue :
    public IceUtil::Shared,
    public IceUtil::RecMutex
{
public:
    ~EncodingResultQueue();

    void result(const EncodingResultPrx&,
        const Ice::ByteSeq&);
    void destroyEncoder(const EncodingResultPrx&);
    void failed(const EncodingResultPrx&, const
        string&);
    void destroy();
private:
    friend class AMI_EncodingResult_resultI;
    friend class AMI_EncodingResult_destroyI;
    void send();

    list<QueueItemPtr> _queue;
};

```

The encoder calls `result` to add an encoded MP3 byte sequence for a particular encoding result proxy. `destroyEncoder` is called to queue a `destroy` invocation, `failed` to queue a `failed` invocation.

tion, and `destroy` to stop sending messages to the client. We wrap each of the queue items in a class called `QueueItem`. `QueueItem` has two sub-classes—one for each type of message that we send.

```
// C++
class QueueItem : public IceUtil::Shared
{
public:
    QueueItem(const EncodingResultPrx&);

    virtual void
    send(const EncodingResultQueuePtr&) = 0;
protected:
    const EncodingResultPrx _result;
};
```

The implementation of the `EncodingResultQueueItem` is as follows:

```
// C++
class EncodingResultQueueItem : public QueueItem
{
public:
    EncodingResultQueueItem(
        const EncodingResultPrx& result,
        const Ice::ByteSeq& encoding) :
        QueueItem(result),
        _encoding(encoding)
    {
    }
    virtual void
    send(const EncodingResultQueuePtr& queue)
    {
        _result->result_async(
            new AMI_EncodingResult_resultI(queue),
            _encoding);
    }
private:
    const Ice::ByteSeq _encoding;
};
```

Next we look at the implementation of the AMI callback:

```
// C++
class AMI_EncodingResult_resultI :
    public AMI_EncodingResult_result
{
public:
    AMI_EncodingResult_resultI(
        const EncodingResultQueuePtr& queue) :
        _queue(queue)
    {
    }
    virtual void
    ice_response()
    {
        _queue->send();
    }

    virtual void
    ice_exception(const Ice::Exception& e)
    {
```

```
        // Error handling
    }

private:
    const EncodingResultQueuePtr _queue;
};
```

As you can see, receipt of the `ice_response` callback prompts the queue to send the next queued item. Here is how a message gets queued:

```
// C++
void
EncodingResultQueue::result(
    const EncodingResultPrx& result,
    const Ice::ByteSeq& encoding)
{
    Lock sync(*this);
    _queue.push_back(
        new EncodingResultQueueItem(
            result, encoding));
    if(_queue.size() == 1)
    {
        _queue.front()->send(this);
    }
}
```

The code creates a new queue item and adds it at the tail of the queue. If this is the only item in the queue, the code sends the item.

Next we look at `send`. Remember that this operation is called by the AMI callback to trigger the sending of the next message in the queue:

```
// C++
void
EncodingResultQueue::send()
{
    Lock sync(*this);
    assert(!_queue.empty());
    _queue.pop_front();
    if(!_queue.empty())
    {
        _queue.front()->send(this);
    }
}
```

Note that we do not dequeue a message until `send` is called by the AMI callback. This ensures that the addition of another message to the queue will not trigger another `send` while an AMI callback is outstanding.

Conclusion

This concludes the implementation of the server side of the application. I did not present the client-side changes but I encourage you to have a look at the [source code](#) to see what is necessary. In the next article in this series, I will further extend the server such that it can store encodings for clients to be picked up at a later date.

Teach Yourself IceGrid in 10 Minutes

Michi Henning, Chief Scientist

Introduction

If you look at the title of this article, your reaction may well be “Ten minutes? That’s ridiculous—no-one can learn IceGrid in that time.” If so, you are right: you *cannot* learn IceGrid in ten minutes, at least not if you want to use the more advanced features of IceGrid. In that case, you will have to put up with the learning curve and spend a fair bit more time than ten minutes (but learning IceGrid is a lot easier than rocket science).

The title simply follows the naming theme of a popular series of books with titles such as “Teach Yourself Linux in 10 minutes”, “Teach Yourself SQL in 10 Minutes”, and many others in the same vein. (In fact, looking at these books, it appears that there is hardly any computing topic that you cannot learn in ten minutes.) Personally, I do have a problem with books that claim to be able to impart any significant amount of information on complex computing topics in a few hours, let alone minutes—but that is a matter for a future editorial instead of this article. Regardless, it *is* possible to get up and running with IceGrid in a few minutes, at least for the basics. To be honest, it will likely take a bit more time than ten minutes, probably more like thirty, but who’s counting...

So, if you have never used IceGrid before, this article is for you: it explains how you can avoid manual endpoint administration and get a server activated on demand when a client invokes an operation on an object in that server. You will be surprised how easy this is—a few simple steps are sufficient to achieve it.

Avoiding Hard-Wired Port Numbers

You will probably have seen server-side code such as the following:

```
// Java
// ...
Ice.ObjectAdapter = communicator().
    createObjectAdapterWithEndpoints(
        "MyAdapter", "tcp -p 10000");
// ...
```

This is the simplest and most straightforward way of creating an object adapter. Unfortunately, it is also one of the most useless:

- The server hard-wires the endpoint information into the source code. As a result, if you want to move the server to a different port for some reason, you will need to recompile the code.

- You need to manually administer the port numbers that are used by servers because no two servers can listen on the same port. If you have a large number of servers, this rapidly becomes tedious.

To improve on this situation, you can pass the port information into the call to `createObjectAdapterWithEndpoints`, for example:

```
// Java
// ...
Ice.ObjectAdapter = communicator().
    createObjectAdapterWithEndpoints(
        "MyAdapter", args[0]);
// ...
```

This code allows you to pass the endpoint specification into the program as a command-line argument. This gets rid of hard-wiring the endpoint into the source code, but is still awkward, for two reasons:

- Ice already has a built-in mechanism for doing exactly the same thing.
- You cannot use IceGrid’s location and server activation features if you create the object adapter in this way.

Here is how to achieve the same thing properly:

```
// Java
// ...
Ice.ObjectAdapter = communicator().
    createObjectAdapter("MyAdapter");
// ...
```

This code is identical, except that it calls `createObjectAdapter` instead of `createObjectAdapterWithEndpoints`. The code does not specify an endpoint for the adapter, so the Ice run time must use some other means to determine what endpoint to use. The implementation of `createObjectAdapter` behaves as follows:

- If the property `MyAdapter.Endpoints` is not set, the run time creates the adapter without endpoints. Obviously, because such an adapter does not listen on any network interface for incoming requests, it is not useful for distributed computing. However, an adapter without endpoints is useful for bidirectional communication and used internally by the Ice run time.
- Otherwise, the run time uses the value of `MyAdapter.Endpoints` to determine at what endpoint(s) the adapter will listen for incoming requests.

With this changed code, we can control the endpoint for the server’s adapter from the command line, for example:

```
$ java MyServer.Main --Ice.Config=config
```

This assumes that the `MyAdapter.Endpoints` property is set in the configuration file `config` as follows:

```
MyAdapter.Endpoints=tcp -p 10000
```

With languages other than Java, you can also set the `ICE_CONFIG` environment variable to the path name of the configuration file instead of using a command-line option—please see the [Ice Manual](#) for details on how to set properties.

With this configuration, the client can construct an initial proxy for an object in the server as usual: as long as the client knows the object identity and the endpoint, it can use a stringified proxy and pass that to `stringToProxy`. With the preceding configuration, assuming that the object identity of an object is `Object1`, the client can use the following stringified proxy to reach the object:

```
Object1:tcp -h somehost.xyz.com -p 10000
```

Using IceGrid's Location Service

By moving the port number that is used by an object adapter out of the source code, we have gained some flexibility because we now can run a server at a different port without having to recompile the code. However, if we have lots of servers, we still need to manually administer which port is used by what server. Moreover, because clients specify the server's port number in their stringified proxy, whenever we change the machine on which a server runs, or the port number at which it listens, we also need to update the configuration of all clients.

Clearly, it would be preferable to not be burdened with all this administrative overhead. Ideally, we want to be able to run servers on arbitrary machines and on arbitrary ports that are dynamically assigned by the operating system, and have clients bind to the servers without any change in configuration.

The location service that is built into IceGrid provides a neat solution for exactly this scenario. The IceGrid location service allows clients to dynamically (and transparently) acquire the current endpoint for a server, regardless of the machine and the port at which a server is running. Similarly, for servers, no ports need be configured. We can run a server on any machine and let the operating system choose a free port for the server, without having to administer anything.

The location service works by replacing the endpoint information in the proxy that is used by a client with a symbolic name, for example:

```
Object1@MyAdapter
```

Such a proxy is known as an *indirect proxy*, because the proxy will be bound to the server endpoint with an extra level of indirection via IceGrid. (In contrast, a proxy that includes a specific endpoint is known as a *direct proxy*.) When the client invokes an operation using an indirect proxy, the client-side run time contacts the IceGrid locator behind the scenes and asks for the machine and port at which `MyAdapter` can be found. If the server is running, the locator knows the endpoint for the adapter and returns that to the client.

Once the client-side run time is aware of the actual endpoint, it then sends the request to the server. The entire process is transparent to application code and quite similar to the way the DNS resolves domain names to IP addresses.

The Ice run time also uses a number of optimizations and caching to prevent this extra level of indirection from becoming a performance bottleneck. Typically, this means that each client will contact the locator only once, the first time it binds to a particular endpoint; future invocations are sent directly to the server without first contacting the locator.

Servers keep the locator up-to-date by contacting it whenever they activate an object adapter: each server updates the locator with its current IP address and port number, so the locator can, in turn, pass that information to clients when they resolve an indirect proxy.

For all this to work, both clients and servers must agree to use the same locator. The location service is provided by the IceGrid registry, so this is the same as saying that clients and servers must agree to use the same registry. To do this, clients and servers must be configured with a single property, `Ice.Default.Locator`. This property specifies the proxy to the IceGrid location service and, if set, enables indirect binding for clients, as well as registration of endpoint details by servers. So, for both clients and servers, we simply need to set this property, for example:

```
Ice.Default.Locator=IceGrid/Locator:tcp -h registryhost.xyz.com -p 12000
```

We can set this property in a configuration file or on the command line for client and server. The proxy states that the locator runs on host `registryhost.xyz.com`, at port 12000, with the object identity `IceGrid/Locator`. (This is the default object identity of the IceGrid locator. You can change this identity by setting the property `IceGrid.InstanceName`—see the [Ice Manual](#) for details.)

To enable indirect binding, we need to run the location service, that is, run the `icegridregistry` process. The registry requires a minimum of configuration:

```
IceGrid.Registry.Client.Endpoints=tcp -p 12000
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=db/registry
```

```
IceGrid.Registry.DynamicRegistration=1
```

The `IceGrid.Registry.Client.Endpoints` property determines the endpoint at which the location service runs. You must configure clients and servers with `Ice.Default.Locator` such that the endpoint matches the locator endpoint.

TEACH YOURSELF ICEGRID IN 10 MINUTES

Note that the proxy you specify with `IceGrid.Registry.Client.Endpoints` *must* be a direct proxy with a fixed port number: it provides the one fixed point that clients and servers need in order to use indirect binding. (The locator proxy cannot be an indirect proxy because that would create a chicken-and-egg problem: to resolve the proxy to the locator, we need a locator, but we cannot find the locator without resolving the proxy...)

You must set the server and internal endpoint properties to one or more protocols, but you need not specify a specific port for these two properties; clients and servers find the actual endpoint by contacting the locator at run time.

The `IceGrid.Registry.Data` property specifies the path name to a directory in which the registry keeps its database.

Finally, you must set `IceGrid.Registry.DynamicRegistration` to a non-zero value. (Without this setting, servers will not be allowed to register their object adapter endpoints unless they have been explicitly deployed. I will return to explicit deployment shortly.)

Having specified these property settings in a file `config.registry`, you can run the registry as follows:

```
icegridregistry --Ice.Config=config.registry
```

For the server, you only need two properties to make the server register its adapter with the locator:

```
MyAdapter.Endpoints=tcp
MyAdapter.AdapterId=MyAdapter
```

Note that `MyAdapter.Endpoints` has changed: it now only specifies a protocol, but no longer specifies a port number. In effect, this says “I want `MyAdapter` to use TCP/IP, but I don’t care about what port number it listens at.” You also must set `MyAdapter.AdapterId`. For example, we could set this property as follows:

```
MyAdapter.AdapterId=FooBar
```

In that case, the proxy used by clients to bind to the server would look like this:

```
Object1@FooBar
```

The `<adapter-name>.AdapterId` property controls three things:

- It tells the server-side run time to register the adapter with the locator.
- It sets the ID by which the adapter is known to the locator and to clients.
- It causes the adapter to produce indirect proxies instead of direct proxies.

The second point is particularly important: it allows two servers to use the same adapter name, such as `MyAdapter`, without causing a naming conflict in the registry: by assigning different adapter IDs to these adapters, they remain distinguishable to the registry and to clients. (Without such a renaming mechanism, all adapters in all servers would have to have unique names, which is difficult to ensure, especially if the servers are written by independent developers.) Adapter IDs are also useful for configuration because they allow tools to unambiguously refer to a specific adapter by its ID.

With these two properties set, once you start the server, the server contacts the registry and informs it of the endpoint details for `MyAdapter` and, when a client uses a proxy such as `Object1@MyAdapter`, it will correctly bind to the server, regardless of what machine the server runs on and at what port number it listens. You can see this magic in action if you set the `Ice.Trace.Location` property on the client side, which shows you the behind-the-scenes activity during binding of indirect proxies.

You can very easily see all of this in action by modifying the demo in `demo/Ice/hello` a little bit. As it stands, this demo uses direct binding so, by converting it to use indirect binding via IceGrid instead, you can see exactly what is involved. Here are the changes that are necessary:

- The server configuration sets the property `Hello.Endpoints` to the value `tcp -p 10000:udp -p 10000:ssl -p 10001`
Change this property to the value `tcp:udp:ssl`
- Add the property setting `Hello.AdapterId=HelloAdapter` to the server configuration.
- The client configuration sets the property `Hello.Proxy` to the value `hello:tcp -p 10000:udp -p 10000:ssl -p 10001`
Change this property to the value `hello@HelloAdapter`
- Set the property `Ice.Default.Locator` to the proxy of the registry in both client and server configuration.

Now, when you run the demo while the registry is running, the client uses indirect binding and the server uses a port that is assigned by the operating system from the ephemeral port range.

Activating Servers Automatically

With the configuration we just discussed, you can start a server and make the server’s endpoint information available via IceGrid without having to manually configure host names and port numbers. However, all this works only for as long as the server is running. Often, this is not a problem: you can simply start the server when the machine boots (by making the appropriate start-up entries in `/etc/rc.d` or the Windows registry); once the server is up, you

TEACH YOURSELF ICEGRID IN 10 MINUTES

can forget about it and let it do its job. However, there are three drawbacks to this:

- Maintaining initialization scripts or registry entries for many servers quickly becomes tedious.
- Servers consume operating system resources even when they are idle.
- Servers may crash or malfunction.

The second point is not too serious—the only thing a server consumes while it is idle is a slot in the process table, a few file descriptors, and swap space, none of which are normally scarce resources. However, the third point deserves more attention because a server can malfunction due to no fault of its own. For example, the operating system can run out of swap space and cause a memory allocation failure in the server. Depending on exactly where the problem occurs, the server code may simply give up and exit or, worse, misbehave in less obvious ways. (And the failure may occur in a third-party library that is used by the server and whose quality you cannot control.) Another scenario for unexpected server death is a system administrator who accidentally kills the wrong process. (Of course, the system administrator will usually in turn blame a buggy script...) The problem with manually started servers is that, well, they are started manually: if a server crashes, it stays down until someone re-starts it.

IceGrid provides a facility to activate servers on demand, when a client first invokes an operation. In a nutshell, automatic server activation is an add-on service to the location service: clients resolve indirect proxies in the usual way; however, if a server is not running at the time a client asks for the server's endpoint, the registry first starts the server and returns the endpoint details to the client once the server has activated its object adapter.

Server activation is taken care of by IceGrid nodes. You must run an IceGrid node on each machine on which you want IceGrid to start servers on demand. In addition, you must run a single IceGrid registry (not necessarily on one of the machines on which you run your application servers). It is the job of each IceGrid node to activate servers on the corresponding machine, to monitor the servers, and to make the servers' status available to the registry.

Frequently, you will run the IceGrid registry on the same machine as one of the IceGrid nodes; because this is a common deployment scenario, IceGrid allows you to combine the registry and a node into a single process by setting the `IceGrid.Node.CollocateRegistry` property to a non-zero value. In addition to the registry properties we used in the preceding section, the node also requires a few configuration properties:

```
# File config.icegrid

# Registry configuration (as before)
IceGrid.Registry.Client.Endpoints=tcp -p 12000
IceGrid.Registry.Server.Endpoints=tcp
IceGrid.Registry.Internal.Endpoints=tcp
IceGrid.Registry.Data=db/registry
```

```
# Only required if you want servers to register
# themselves without explicit deployment. If all
# servers are deployed explicitly, this property
# can be left unset.
IceGrid.Registry.DynamicRegistration=1
```

```
# Node configuration
IceGrid.Node.CollocateRegistry=1
IceGrid.Node.Name=node1
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Data=db/node
```

```
# Set the default locator so the node and admin
# tools can find the registry.
Ice.Default.Locator=IceGrid/Locator:tcp
-h registryhost.xyz.com -p 12000
```

The additional node configuration sets

```
IceGrid.Node.CollocateRegistry to indicate that the node
should also act as a registry. The IceGrid.Node.Name property
assigns a symbolic name to the node. This name can be anything—
it serves to distinguish nodes that use the same registry, that is, the
nodes of a registry must have unique names. The
IceGrid.Node.Data property sets the path name of a directory
in which the node stores information about its servers.
```

Now we can start a node that also includes a registry:

```
icegridnode --Ice.Config=config.icegrid
```

On the client side, no changes are required to make the client work with automatically activated servers because all the work is done by the registry. To make the server work with automatic activation, we must make two changes:

- update the server configuration
- deploy the server

The first point is taken care of very easily: the server now requires no configuration at all, other than the setting of `Ice.Default.Locator`. In particular, we no longer need to specify an endpoint or an adapter ID because, as we will see in a moment, that configuration shifts from the server to the server's deployment.

To get IceGrid to activate the server on demand, we need to inform IceGrid of the particulars of the server. Here are the essential items of information that IceGrid needs to know so it can start the server:

- an application name
- a node name
- a server identifier
- the path name to the executable of the server
- the name of the server's adapter
- the protocol to be used by the server

IceGrid expects these items to be presented in a deployment descriptor. Deployment descriptors are written in XML. Here is the deployment descriptor for our example:

```
<!-- demo.xml -->
<icegrid>
  <application name="demo">
    <node name="node1">
      <server id="DemoServer"
        exe="/usr/bin/demoserver"
        activation="on-demand">
        <adapter name="MyAdapter"
          endpoints="tcp"/>
      </server>
    </node>
  </application>
</icegrid>
```

Much of this is self-explanatory. All of the information is presented as sub-elements of the `icegrid` element.

- The application name identifies the deployment information for an application (which may have more than one server). In other words, the application name serves as a convenient handle when we need to identify a particular deployment (as we will in a moment when we use the `icegridadmin` tool).
- The node name identifies the machine on which the server will execute or, more precisely, it provides the name of the node that will be instructed to start the server—the server will execute on the machine that runs the node with that name. The node name is the same name that we configured earlier by setting the `IceGrid.Node.Name` property.
- The server ID is a label that identifies the server. It allows us to refer to a particular server by name, for example, to enquire about the server's status with an administrative tool.
- The `exe` attribute provides the path name of the server's executable. (Usually, you will use an absolute path name here because relative pathnames are interpreted relative to the node's working directory.)
- The `activation` attribute specifies that the server should be activated on demand, when a client invokes an operation on one of the server's objects. (IceGrid provides a number of other activation modes—please see the [Ice Manual](#) for details.)
- The `adapter` element's `name` attribute must specify the adapter name that is used by the server. (You can also optionally set an `id` attribute; if that attribute is not set, the default adapter ID is `<server-ID>.<adapter-name>`.)
- The `endpoints` attribute specifies the protocol(s) to be used by the server.

Now that we have a deployment descriptor, we can deploy the application, that is, inform the IceGrid registry about these details:

```
icegridadmin --Ice.Config=config.icegrid -e
'application add demo.xml'
```

The `-e` option tells `icegridadmin` to execute the commands provided as the option argument. In this case, the `add` command tells the tool that we want to add the information in `demo.xml` to the registry database. Note that the command also points the tool at the configuration file for the registry and the node. The only property setting that is read by `icegridadmin` is `Ice.Default.Locator`, which the tool needs so it knows how to contact the location service.

This is all that is necessary to have your server activated on demand. Provided that you have deployed the server with the registry, it now starts automatically as soon as the first client tries to contact an object in that server.

Other Features

This article only covers the basics of IceGrid to get you started. There are many other features in IceGrid, some quite sophisticated, such as replication and load balancing, allocation of particular servers for exclusive use by clients, and templates to simplify deployment and configuration of large numbers of servers. You can also arrange for a server to stop automatically once it has been idle for some time, to conserve machine resources. You can even arrange for software updates to be downloaded to a number of remote machines, allowing you to automatically update application software from a central point without intervention at the remote end. As usual, please consult the [Ice Manual](#) for more information on these features, as well as Matthew Newhook's articles on IceGrid in this and previous issues of *Connections*.

Summary

IceGrid makes it very easy to get away from manual port administration and, through indirect binding, allows you to move servers from one machine to another (for example, to balance machine load) without having to update the configuration of all deployed clients. In addition, the central administration of IceGrid allows you to deploy a large number of servers easily and efficiently, without getting overwhelmed by lots of detail.

If you want to experiment with IceGrid, I suggest you start with the demo that is provided in the `demo/IceGrid/simple` directory in the Ice distribution. This article was inspired by that demo, so you should have no problems getting started. But, please, give yourself just a little more than ten minutes...

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: Why is there no `Ice.Exception` base class in Ice for Java?

In Ice for C++ (and other language mappings), all Ice exceptions derive from a common base class. For example, in C++, we have `Ice::Exception` at the root, with derived classes `Ice::LocalException` and `Ice::UserException`. In turn, all the Ice run-time exceptions derive from `LocalException`, and all the Ice user exceptions derive from `UserException`.

The advantage of having a common base class for user- and run-time exceptions is that you can catch all Ice exceptions with a single exception handler:

```
// C++
try
{
    someProxy->someOp();
}
catch(const Ice::Exception& ex)
{
    cerr << ex;
}
```

In Java, we also have `Ice.LocalException` and `Ice.UserException`, but these two classes are *not* derived from a common base class. If you want to catch all Ice exceptions, you must write two separate exception handlers:

```
// Java
try
{
    someProxy.someOp();
}
catch(Ice.UserException ex)
{
    System.out.write(ex);
}
catch(Ice.LocalException ex)
{
    System.out.write(ex);
}
```

So, why this difference? The reason is Java's checked exception model. (People either hate or love this model—[Kevlin Henney](#) (among many others) provides an [interesting discussion](#) of its trade-offs.)

Java distinguishes between two kinds of exceptions, checked exceptions and unchecked ones. For checked exceptions, the language, at compile time, enforces that a method must declare all checked exceptions that it can possibly throw in a separate `throws` clause. This includes any exceptions that (recursively) might be thrown by any called methods. On the other hand, for unchecked exceptions (which are exceptions that derive from `java.lang.RuntimeException` or `java.lang.Error`), the language does not require you to list them explicitly in a `throws` clause—any method can throw an unchecked exception at any time.

Unchecked exceptions were added to the language out of necessity. For example, imagine the consequences of `NullPointerException` being a checked exception: either every method would need a `throws` clause for this exception, or the body of every method would have to catch and swallow this exception (or translate it to some other exception that can be thrown). Clearly, this would be quite intrusive and messy.

Ice follows the Java philosophy: Ice run-time exceptions are unchecked exceptions and Ice user exceptions are checked exceptions. This allows you to write code without eternally having to write `throws` clauses for Ice run-time exceptions, while still enforcing that your methods correctly deal with user exceptions. The down-side of this approach that you need two exception handlers if you want to catch both Ice user- and run-time exceptions.

Note that you *can* catch all Ice exceptions with a single exception handler:

```
// Java
try
{
    someProxy.someOp();
}
catch(java.lang.Throwable ex)
{
    // Catches too much...
}
```

Sure enough, this catches all Ice user- and run-time exceptions, but the glitch is that it catches everything else as well, including exceptions that have nothing to do with Ice. As a work-around, you could add further processing in the handler to determine whether the exception is not an Ice exception and, if so, rethrow it:

```
// Java
static void
throwIfNonIceException(java.lang.Throwable ex)
{
    if(!(ex instanceof Ice.LocalException) &&
        !(ex instanceof Ice.UserException))
    {
        throw ex;
    }
}
```

```

    }
}
try
{
    someProxy.someOp();
}
catch(java.lang.Throwable ex)
{
    throwIfNonIceException(ex);
    // Handle Ice exception...
}

```

However, most people would agree that this rather obscures the issue; it is clearer and simpler to write two separate exception handlers in the few places in the code where you need to catch both Ice user- and run-time exceptions.

Q: How do I run the clients and servers on different hosts?

By default, the demo programs that ship with Ice assume that you will run client and server on the same host. This behavior is controlled by two configuration files, `config.client` and `config.server`. For example, here is the relevant line for the server configuration of the hello demo:

```
Hello.Endpoints=tcp -p 10000:udp -p 10000:ssl -p 10001
```

This says that the server's object adapter named `Hello` will listen for incoming requests on port 10000 for UDP and TCP, and on port 10001 for SSL. Because this configuration does not use the `-h` option to explicitly specify an interface, the server binds itself to all network interfaces on its machine.

The corresponding entry for client looks like this:

```
Hello.Proxy=hello:tcp -p 10000:udp -p 10000:ssl -p 10001
```

This configures the proxy that is used by the client to make an invocation. Again, because the configuration does not use the `-h` option, the client will try all network interfaces on its machine when it tries to reach the server.

If you want to run client and server on different machines, you need to modify the configuration of the client to specify the server's machine. For example, if the server runs on host `www.zeroc.com`, you can modify the client configuration to specify that machine:

```
Hello.Proxy=hello:tcp -h www.zeroc.com -p 10000:
udp -h www.zeroc.com -p 10000:ssl -h www.zeroc.com
-p 10001
```

This configuration assumes that the DNS for the client can correctly resolve the domain name `www.zeroc.com`; if that is not the case, you can also use an IP address instead of a domain name.

Because you can configure a separate endpoint for each protocol, you can also create more complex configurations. For example, for a machine with separate interfaces for an external network and an internal network, you could use a server configuration as follows:

```
Hello.Endpoints=tcp -h internal.zeroc.com -p
10000:udp -h internal.zeroc.com -p 7859:ssl -h
external.zeroc.com -p 10001
```

With this endpoint specification, the server will accept TCP and UDP requests only on the internal interface, at ports 10000 and 7859, respectively, and will accept SSL requests only on the external interface, at port 10001.

Regardless of what configuration you use, if a client cannot reach the server, the first thing to do is to run both client and server with `--Ice.Trace.Network=2` and check that the endpoint that the client tries to connect to matches the endpoint at which the server listens. If not, the fault is inevitably in the endpoint configuration of either client or server (or both): the configurations must match for the client to be able to reach the server.

Instead of configuring endpoints manually, you can also let IceGrid take care of port allocation for you. Please check the [Ice Manual](#) for details.