Let's Chat! (Part 1)

José Gutíerrez de la Concha, Software Developer
Michi Henning, Chief Scientist

In this series of articles, we will guide you through the process of designing and implementing a complete application with Ice. The application is a chat application that allows users to connect to a chat room and exchange messages with other users. This application is simple enough to discuss in detail and also presents a number of interesting design challenges that are encountered by distributed applications in general. In other words, a chat application makes for an excellent case study.

Requirements

Before we launch into the actual design of the application, let's draw up a list of requirements. These requirements are typical for many distributed applications and, therefore, create design constraints that are similar to those of many real-world projects. So, here are our high-level requirements:

- The chat application is to be implemented using a typical client–server architecture: clients send messages to a central server that, in turn, makes these messages available to other clients.
- The chat server must require no or only minimal administration.
- Traffic between clients and server must be secure, so privacy is protected even when clients and server communicate via an insecure public network.
- The application must work if clients and servers are protected by firewalls and clients must be able to use the application without any changes to the client-side network or firewall configuration.
- Clients will be implemented in a variety of languages and run on various platforms. In particular, it must be possible to use a web browser as a client.
- Clients may need to run in bandwidth-constrained environments, so network traffic must be minimized as much as possible.

The application supports only a single chat room. (Extending the application to multiple chat rooms does not present any difficulty but would increase code size beyond what is reasonable for this series of articles.)

Also note that, for the remainder of this article, we will use the term "chat client" to mean the application on a user's computer that allows the user to send messages and to view messages from other users. The term "chat server" refers to the part of the application that distributes messages to chat clients. The chat server does not have any user interface.

Design

For this series of articles, we will show you the design and implementation of the following clients. (We cover the server and C++ client implementation in this installment, and present the other clients in the next installment.)

- C++ command line client
- Java Swing GUI client
- .NET WPF (Windows Presentation Foundation) client
- PHP web page client
- Silverlight web page clients

Note that, as of Ice 3.3, we provide only a client-side run time for PHP and Ruby, but no server-side run time. Obviously, this means that we cannot write our chat server using these technologies. However, the absence of a server-side run time also has implications on how we must design our interfaces. To see why, recall that the terms *client* and *server* have meaning only within the context of a particular invocation: clients are active entities that send requests, and servers are passive entities that respond to requests. It follows that, for platforms where we do not provide a server-side run time, it becomes impossible to respond to a request.

This matters for our chat application because we must consider how messages get from the chat server to connected clients. If you recall Matthew Newhook's article "Highly Available IceStorm" in Issue 28 of *Connections*, we can choose between two different communication models to distribute messages to clients:

- **Push Model**: Clients provide an Ice object with an operation that the chat server invokes to deliver messages to clients. In this model, the chat server acts as the client and the chat client acts as the server, that is, while the chat server delivers messages to clients, the normal roles are reversed.
- **Pull Model**: Clients invoke an operation on an object provided by the chat server to retrieve messages that are sent by other users. In this model, chat clients and server are "pure" clients and server because they exclusively act in the client or server role, respectively.

The push model is comparable to interrupt handling, where chat clients display messages as they are sent by the server, whereas the pull model requires polling: clients must continuously contact the server just in case a message is ready.

Back to our chat clients… Clients both originate messages (when a user types a message) and receive messages (when a client displays a message typed by another user). To originate messages, a client can simply invoke an operation on an object provided by the chat server. However, to receive messages, clients can use the push model only for those environments that provide the Ice server-side run time. Specifically, PHP clients cannot use the push model.

It follows that, for PHP clients, we have no option but use the pull model whereas, for C++, Java, .NET, and Silverlight clients we can use either model. The question is, which one should we use?

In general, the push model is both easier to implement and easier to scale. For one, the push model is stateless: the server can pass each incoming message immediately to all connected clients. In contrast, the pull model requires the server to store messages and buffer them until each of the connected clients has pulled that message. Another potential problem with the pull model is the number of incoming connections and the number of threads that are required. We will discuss these issues in more detail when we examine our design for the pull model. For now, let us examine the push model, which we will use to implement clients in C++, Java, .NET, and Silverlight.

Note that Ice for Silverlight 0.2 did not offer support for a server-side run time, therefore the initial version of the chat application included a Silverlight client that used the pull model. Now that Ice for Silverlight 0.3 has introduced a limited server-side run time for use with a Glacier2 router, we have updated the chat application to include a Silverlight push client implementation. Both of the Silverlight clients are discussed in the next installment of this article.

Push Model Definitions

To begin with, let us examine the Slice definitions we need to support the push model. Each client provides an Ice object of type `ChatRoomCallback` to the server. The server invokes operations on this object to inform the client of various events:

```
// Slice
module Chat
{
// Implemented by clients
interface ChatRoomCallback
{
    ["ami"] void init(Ice::StringSeq users);
    ["ami"] void join(long timestamp, string name);
    ["ami"] void leave(long timestamp, string name);
    ["ami"] void send(long timestamp, string name, string message);
};
};
```

The operations are called by the chat server as follows:

- The chat server invokes the `init` operation when a client first connects to the chat room. The `users` parameter informs the client of the users that are currently connected to the room.
- The chat server invokes the `join` operation when a new client connects to the chat room.
- The chat server invokes the `leave` operation when a client disconnects from the chat room.

- The chat server invokes the `send` operation whenever a client sends a message to the room.

The `init`, `join`, and `leave` operations allow each client to maintain an accurate picture of who is currently connected to the room, and the `send` operation allows each client to display the messages that are sent to the room. The `timestamp` and `name` parameters allow each client to display time stamps and names, so the user can see when a message was sent and by whom.

Note that this design models asynchronous events (such as a user joining or leaving a room) with operations: each operation corresponds to a different event, and the parameters of each operation correspond to the data for each specific event.

All operations are marked with the `["ami"]` metadata directive so the server can invoke the callback operations asynchronously. This is necessary so the server is protected against misbehaved clients: if a client's implementation of a callback operation blocks for an extended period, the server does not lose a thread of control for the duration of the call.

Cooperating with Firewalls

Chat clients will most often be situated behind a firewall that prevents incoming connections and, more often than not, will use a network that has a different address than the external network, with a router providing NAT (network address translation). This presents a challenge for our push model because, normally, the client side of an Ice request establishes the connection to the server. With the push model, the chat server acts as the client; however, if our chat clients sit behind a firewall that disallows incoming connections, establishing that connection is impossible. (Recall that our requirements do not allow opening a hole in the client-side firewall that would permit the server to connect to the client.)

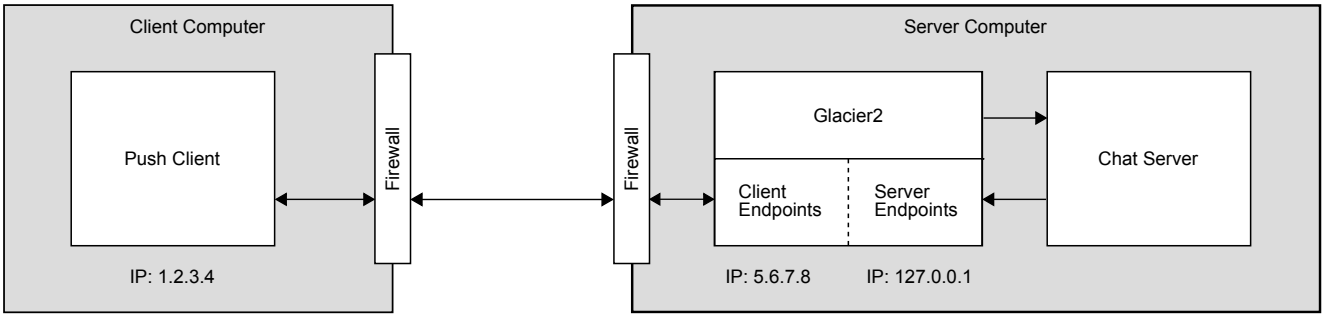
To allow us to use the push model even in the presence of client-side firewalls and NAT, we need to be able to somehow get a callback from the server to the client without having to open a separate connection from server to client. One way to achieve this is to use the bidirectional connection feature of Ice. Bidirectional connections permit a server to invoke a callback in a client over the same connection that was originally opened by the client to the server. This works fine but, as we will see, bidirectional connections help only with part of our design requirements. In particular, our application also requires session management in order to avoid leaking resources in the server. (We will return to the reasons for this shortly.)

As it turns out, Glacier2 offers a pre-built solution that nicely fits our design requirements. Glacier2 acts as a front end for servers: clients connect to Glacier2 instead of the actual server, and Glacier2 forwards the requests on behalf of the clients. This allows Ice applications to co-exist with firewalls in a way that is elegant and flexible, and that requires minimal programming effort. (See Michi's article "Glacier2 in 10 Minutes" in Issue 22 of *Connections* for a quick introduction to Glacier2; you may also want to review the Glacier2 demos that are included in the Ice distribution.) Briefly, Glacier2 provides the following features:

- Glacier2 supports a session concept that allows clients to establish a session with Glacier2. The API supports authentication, with hooks that applications can use to implement custom session creation and authentication.
- A single instance of Glacier2 can forward requests for an arbitrary number of clients and servers. The server-side firewall must accept incoming connections on only a single port, regardless of the number of servers.
- Servers can invoke operations on callback objects provided by clients even if a client-side firewall disallows incoming connections. (Glacier2 uses bidirectional connections for this feature.)

The third point is important in the context of our chat application because it means that clients can use the push model even if they are behind a firewall. To communicate with a server via Glacier2, clients establish a session with Glacier2. In turn, this opens a connection from the client to Glacier2, and Glacier2 forwards client invocations to the target server via a second connection, from Glacier2 to the server. When the server invokes a callback operation on an object provided by a client, instead of opening a separate connection to the client, the server transparently sends the invocation to Glacier2, and Glacier2 forwards the invocation to the client via the same connection that the client established when it created its Glacier2 session. This allows the server to invoke callback operations in the client as long as the client maintains its session with Glacier2. Figure 1 illustrates how push clients connect via Glacier2.

Figure 1: Push client connecting via Glacier2



Because Glacier2's session concept is connection-oriented, servers can call back to the client only while the client's connection with Glacier2 remains open (or, more accurately, only while the client maintains an active session with Glacier2). In other words, if the client loses its connection to Glacier2, Glacier2 automatically destroys the session. To prevent the client's connection with Glacier2 from being closed unintentionally, clients must disable ACM (automatic connection management)—see the Ice Manual for details. In addition, Glacier2 is usually configured to time out sessions that have been idle for some time. To avoid Glacier2 destroying sessions if the chat room is inactive for longer than this, clients must ensure that they periodically cause some activity, for example, by calling `ice_ping`, which resets Glacier2's session timeout.

We will return to Glacier2's session creation and authentication shortly. For now, let us look at the session itself. Chat clients communicate with the chat server via a `ChatSession` interface that is provided by the chat server. The `ChatSession` interface derives from `Glacier2::Session`:

```
// Slice
module Chat
{
exception InvalidMessageException
{
    string reason;
};

interface ChatSession extends Glacier2::Session
{
    void setCallback(ChatRoomCallback* cb);
    ["ami"] long send(string message) throws InvalidMessageException;
};
};
```

The client uses the `setCallback` operation to pass the proxy to its `ChatRoomCallback` object to the server. As we saw previously, the chat server uses this object to push events to the client, and `setCallback` provides this object to the server.

The client invokes the `send` operation to send a message to the chat server. The server, in turn, passes the message to all connected clients by invoking the `send` operation on their respective `ChatRoomCallback` objects. The `send` operation can throw an `InvalidMessageException`, for example, if a message contains characters we want to disallow, or if the message exceeds a size limit.

Note that `send` is marked as an AMI operation, so clients can call `send` asynchronously to avoid blocking. (As of Ice 3.3.0, asynchronous invocations are guaranteed not to block.) This is important for clients that must avoid blocking their UI thread. The `send` operation returns the time at which the server received the message. The client can use this time stamp to label the message for display.

This completes the interface provided to clients by the chat server and, as far as the push model is concerned, this is all that is needed: clients send messages to the chat server by calling `send` on the `ChatSession`, and the chat server distributes messages to clients by calling `send` on each client's `ChatRoomCallback` object.

Pull Model Definitions

Unfortunately, Glacier2's session concept is of no use at all when it comes to making callbacks from the chat server to a PHP client, or to a Silverlight client using Ice for Silverlight 0.2 or earlier, because these clients do not have a server-side Ice run time and, therefore, cannot implement Ice objects. To allow our application to work with such clients, we must abandon the push model and resort to the pull model. Before launching into the design of the interfaces however, we want to discuss a number of interesting design issues around the pull model.

Designing a Pull Model

For the pull model, clients invoke an operation on the chat server in order to receive messages and changes to the membership of the chat room. This means that, in outline, the server must provide an operation such as the following:

```
// Slice
ChatRoomEventSeq getUpdates();
```

A `ChatRoomEventSeq` is a sequence of updates. (We can ignore the exact definition of the sequence elements for the moment.) The basic idea is that a client periodically calls `getUpdates`. Each call returns whatever state changes have accumulated since the last time the same client called `getUpdates`. One immediate consequence of this design is that the server must buffer updates for each connected client, either in memory or in a database. Either way, doing this will be more complex to implement than the push model, which allows the server to get rid of each update immediately.

Another consequence of the pull model is that it potentially makes the server vulnerable to misbehaved clients: if a connected client fails to call `getUpdates`, the server will accumulate more and more state on behalf of that client; if the server does not have a way to discard that state, it will eventually run out of memory or disk space. We will return to how the server can deal with this situation when we discuss the session design of our application. For now, let us consider a different question: "What should happen if there are no updates ready at the time the client calls `getUpdates`?" Answering this seemingly innocuous question is not as simple as one might expect.

Basically, we have two choices for the behavior of `getUpdates` if it is called at a time when no updates are ready:

- Block the caller inside `getUpdates` until an update becomes ready so that the returned sequence always contains at least one element.
- Return immediately with an empty sequence.

Impulsively, we might decide that busy-waiting with a non-blocking `getUpdates` is even worse than blocking because, potentially, clients will incur lots of network traffic by continuously asking for updates when none are ready. So, let us examine some of the consequences of a blocking pull model.

One issue that immediately arises is the question of how to implement a blocking `getUpdates` in the server. If we naïvely implement `getUpdates` as an ordinary synchronous operation, we will use up one thread in the server for each client that blocks in the operation, up to the limit of the server's thread pool. If we have, say, two thousand connected clients, clearly this will not work because the server is not going to have enough memory for that many threads.

So, we would need to limit the server's thread pool to something less, say a few dozen threads. Doing this will work even if there are more clients than we have threads in the thread pool because, once all available threads in the server are in use, incoming requests from other clients will simply queue up in the TCP/IP transport buffers, to be processed once a thread becomes available. However, we would need separate thread pools to ensure that operations other than `getUpdates` could be processed concurrently. In addition, threads are an expensive resource, particularly with respect to memory (due to the need to allocate a separate stack for each thread).

In order to reduce the number of threads, we could implement a blocking `getUpdates` in the server with AMD (asynchronous method dispatch). AMD allows the server to indefinitely block an arbitrary number of requests without using a separate thread for each request. One disadvantage of AMD, however, is that the implementation is somewhat more complex.

Another issue with a blocking pull model relates to the way clients interact with the server. Chat clients typically will have a GUI, and most GUI libraries require use of a dedicated UI thread to keep the GUI up to date. This means that if clients make blocking calls from the UI thread, the user interface becomes non-responsive. So, a blocking

pull model would in fact not be convenient for clients because, in order to avoid freezing the UI, clients would have to invoke the `getUpdates` operation asynchronously anyway. In other words, all the effort in the server to implement a blocking `getUpdates` would be wasted because clients will use asynchronous invocations regardless. (This is true in general—if a client wants to avoid blocking in a remote invocation, the client must use AMI because a synchronous invocation can block the client even if it does not block in the server, for example, during connection establishment.)

The Ice for Silverlight HTTP bridge, via which the web server forwards requests to the target Ice server, currently does not use asynchronous invocations to forward requests to the server. Similarly, PHP clients make synchronous calls from the web server to the target Ice server. If we use a blocking `getUpdates` operation, this means that each blocked client request would consume a thread in the web server for the duration of the call. Again, this is undesirable because it does not scale to large numbers of clients.

A blocking `getUpdates` also raises semantic issues. In particular, what should happen if a client wants to leave the chat room while its call to `getUpdates` is currently blocked waiting for updates to arrive? Should `getUpdates` return an empty sequence to indicate that it terminated early or raise an exception? How would the server realize that a client who leaves the chat room also has a request blocked in `getUpdates` and arrange for `getUpdates` to unblock? To be sure, these questions can be answered. But implementing the necessary functionality would considerably complicate the implementation in the server. In short, a blocking `getUpdates` is problematic, so let us turn to the non-blocking version.

With a non-blocking `getUpdates`, the server immediately returns an empty sequence if no updates are ready at the time the client calls the operation. The immediate advantage of this approach is that we do not consume a thread in the server for any length of time. However, the non-blocking pull model is also not without its own set of problems, particularly if we consider malicious clients:

- A naïve client might continuously call `getUpdates` in a tight loop, which incurs unnecessary network traffic and causes the server to do work for no gain.
- A large number of malicious clients might continuously call `getUpdates` (or, in fact, the `send` operation) in an attempt to bring down the server.

The obvious strategy to deal with this concern is to limit the rate at which clients call `getUpdates`. Well-behaved clients can be written to call `getUpdates` once every five seconds or so, so the network and the server do not get flooded with requests.

To prevent clients from sending unreasonable amounts of data with the `send` operation, the server can set its `Ice.MessageSizeMax` property. Note however that our `init` operation can potentially return a large amount of data if there are many users in the chat room, so we cannot set `Ice.MessageSizeMax` smaller than, say, a few kilobytes.

Dealing with malicious clients is harder and requires action at several levels. Here are a few suggestions that, in combination, can be very effective:

- Clients should be required to provide a unique user name and password. Moreover, if a user attempts to join the chat room a second time, the second attempt should automatically cancel the previous login. This prevents a malicious user from starting processes on many different machines for the same user, in an attempt to increase the load on the server.
- Many firewalls allow you to limit the rate at which clients can send messages to a particular port. Setting such a limit makes it more difficult for clients to flood a server.
- Monitor incoming messages in the server for frequency and content. For example, you can scan messages for suspicious characters, such as control characters. If a client sends too many message or messages that appear to be garbage, terminate the client's session.
- Limit the size of incoming messages to something reasonable, say, a few hundred bytes to prevent clients from sending multi-kilobyte messages to the room.

Note that, for a service that is accessible to arbitrary and anonymous users, it is impossible to completely protect a server against denial-of-service attacks because the act of rejecting an incoming request requires effort by the server. If enough malicious clients continuously attack the server, at the very least, legitimate clients will experience a deg-

radation of service. You can mitigate this by allowing only authorized clients with a valid user name and password (or SSL certificate) to contact the server. The assumption is that authorized clients are not malicious. (See the Ice Manual for more information on SSL certificates and configuration.)

Pull Model Interfaces

Recall that, for the push model, we modeled events as operations, with the operation parameters carrying the data for each type of event. Because the pull model disallows operation invocations to deliver events, we now model events as classes, with data members of each class providing the data specific to each event:

```
// Slice
module PollingChat
{
class ChatRoomEvent
{
    long timestamp;
    string name;
};
sequence<ChatRoomEvent> ChatRoomEventSeq;

class UserJoinedEvent extends ChatRoomEvent
{
};

class UserLeftEvent extends ChatRoomEvent
{
};

class MessageEvent extends ChatRoomEvent
{
    string message;
};
};
```

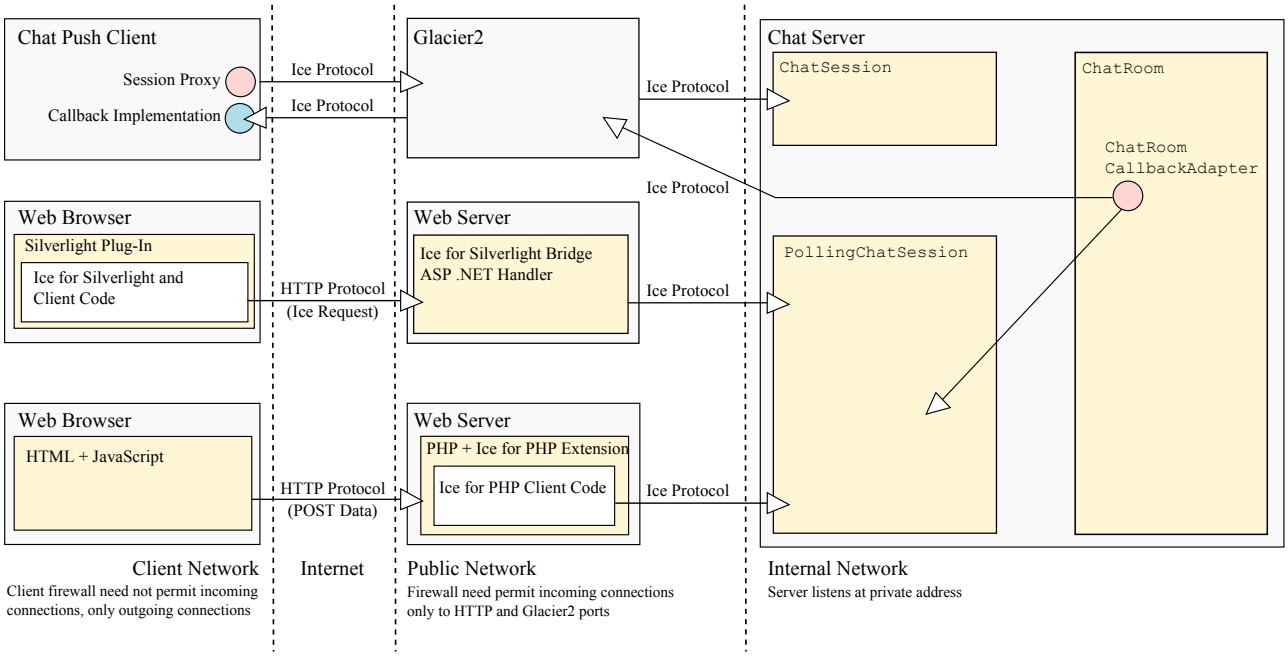
We have three concrete types of event in this definition. The `UserJoinedEvent` and `UserLeftEvent` classes correspond to the `join` and `leave` operations of the push model. The `MessageEvent` class informs the chat client of a new message that was sent to the room. Note that all events derive from a common base class `ChatRoomEvent`, which carries a time stamp and the name of the user. These two data items are common to all events; when a chat client receives a `ChatRoomEvent`, it can use a down-cast to determine which type of event it has received. (`ChatRoomEvent` itself is never instantiated—we treat it as an abstract base class.)

Here is a version of the chat room session for polling clients:

```
// Slice
module PollingChat
{
interface PollingChatSession
{
    ["ami"] Ice::StringSeq getInitialUsers();
    ["ami"] ChatRoomEventSeq getUpdates();
    ["ami"] long send(string message) throws Chat::InvalidMessageException;
    ["ami"] void destroy();
};
};
```

Note that this interface does not derive from `Glacier2::Session`. This is because the invocations on the chat server originate from a web server (a PHP script or the Silverlight bridge) which normally will be behind a firewall anyway, so there is no need for Glacier2 in this case. Of course, this also means that clients connecting from an outside network via Glacier2 must use the push model. The pull model is available only to clients that connect to the chat server via a web server, meaning Ice for PHP clients and Ice for Silverlight clients that use the HTTP bridge. But, seeing that the push model is superior anyway, that is no loss. (See Figure 2 for an overview of this design.)

Figure 2: Application structure



The `getInitialUsers` operation returns the users currently in the chat room. Pull clients call this operation to initialize themselves after joining the room.

The main operation of interest is `getUpdates`. This is the polling operation that clients call to find out what goes on in the chat room. A client calls `getUpdates` whenever it wants to receive one or more events. The server returns a sequence of events that reflect the activity in the room since the previous call to `getUpdates` by the client: if a user sent a message, the sequence contains a `MessageEvent`; if a user joined or left the chat room, the sequence contains a corresponding `UserJoinedEvent` or `UserLeftEvent`.

As for the push model, the client calls `send` to send a message to the chat server. The `send` operation returns the time at which the server received the message. The client can use this time stamp to label the message for display.

The `destroy` operation allows the client to destroy its session, thereby leaving the chat room. (This operation is present in the push model as well and serves the same purpose there; for the push model, the `destroy` operation is provided by the `Glacier2::Session` base interface.)

As for the push model, we use an `["ami"]` metadata directive so clients can use asynchronous invocations to avoid blocking the UI thread.

Session Creation

For the push model, we will use Glacier2 sessions and, therefore, use the mechanism provided by Glacier2 for session creation. However, for the pull model, we need to come up with a session creation mechanism of our own, so polling clients can join the chat room:

```
// Slice
module PollingChat
{
exception CannotCreateSessionException
{
    string reason;
};
```

```
interface PollingChatSessionFactory
{
    ["ami"] PollingChatSession* create(string name, string password)
                                    throws CannotCreateSessionException;
};
};
```

The `PollingChatSessionFactory` is provided by the chat server. To join the chat room, clients call `create`, passing their name and a password. Assuming the server allows the client to join the room, the client receives a proxy to a `PollingChatSession` that the server creates on behalf of that client. The client then calls `getInitialUsers` to populate its list of users, followed by periodic calls to `getUpdates`. Each call informs the client of any state changes to the room since the previous call. To leave the room, the client calls `destroy` on the session, which allows the server to clean up any resources with that client (such as events it has buffered for the client's next call to `getUpdates`).

Server Implementation

Before we look at the various clients, let's discuss how to implement the server. For this article, we chose to implement the server in C++. Figure 2 illustrates the overall structure of our chat application.

Note the structure of the server: the `ChatRoom` class implements most of the application logic. In order to support both push and pull models, the server also implements the `ChatSession` and `PollingChatSession` classes. To deliver messages to clients, `ChatRoom` invokes a `send` function on a `ChatRoomCallbackAdapter` object. This object hides the differences between the push and poll models and interfaces with `ChatSession` and `PollingChatSession` as appropriate. (We will examine this object in more detail shortly.)

Implementing ***`ChatRoom`***

Sessions communicate with the `ChatRoom` using normal C++ function calls, that is, `ChatRoom` is an ordinary C++ object, not a servant. Here is the class definition:

```
// C++
class ChatRoomCallbackAdapter { /* ... */ };
typedef IceUtil::Handle<ChatRoomCallbackAdapter> ChatRoomCallbackAdapterPtr;

class ChatRoom : public IceUtil::Shared
{
public:
    void reserve(const string&);
    void unreserve(const string&);
    void join(const string&, const ChatRoomCallbackAdapterPtr&);
    void leave(const string&);
    Ice::Long send(const string&, const string&);

private:
    typedef map<string, ChatRoomCallbackAdapterPtr> ChatRoomCallbackMap;

    ChatRoomCallbackMap _members;
    set<string> _reserved;
    IceUtil::Mutex _mutex;
};
typedef IceUtil::Handle<ChatRoom> ChatRoomPtr;
```

The code defines `ChatRoomPtr` as a smart pointer to a chat room, so we do not have to worry about memory deallocation chores. (See "Who's Counting?" in Issue 25 of *Connections* for more detail on smart pointers.)

The `_reserved` member is a set of strings that stores the names of the users that have created a session but have not yet joined the chat room. In contrast, the `_members` map stores the users currently in the room, that is, the users that have called `join`. `_members` is a map that maps the user name to the callback object that receives notifications about events. As shown in Figure 2, even for the pull model, there is such a callback object—for the pull model, the session implementation provides this object. We will see the purpose of the `ChatRoomCallbackAdapter` class in a moment.

The `reserve` and `unreserve` member functions maintain the `_reserved` set, while the `_mutex` member synchronizes access to member variables.

```
// C++
void
ChatRoom::reserve(const string& name)
{
    IceUtil::Mutex::Lock sync(_mutex);
    if(_reserved.find(name) != _reserved.end() || _members.find(name) != _members.end())
    {
        throw string("The name " + name + " is already in use.");
    }
    _reserved.insert(name);
}

void
ChatRoom::unreserve(const string& name)
{
    IceUtil::Mutex::Lock sync(_mutex);
    _reserved.erase(name);
}
```

Note that `reserve` checks both `_reserved` and `_members` and allows a name to be reserved only if it does not appear in either member, that is, the name must not have been used to create a session, and must not belong to any user currently in the chat room.

The `join` operation adds a user to a chat room:

```
// C++
void
ChatRoom::join(const string& name, const ChatRoomCallbackAdapterPtr& callback)
{
    IceUtil::Mutex::Lock sync(_mutex);
    IceUtil::Int64 timestamp = IceUtil::Time::now().toMilliSeconds();
    _reserved.erase(name);

    Ice::StringSeq names;
    ChatRoomCallbackMap::const_iterator q;
    for(q = _members.begin(); q != _members.end(); ++q)
    {
        names.push_back((*q).first);
    }

    callback->init(names);

    _members[name] = callback;

    UserJoinedEventPtr e = new UserJoinedEvent(timestamp, name);
    for(q = _members.begin(); q != _members.end(); ++q)
    {
        q->second->join(e);
    }
}
```

The implemention creates a time stamp and then removes the user name from the `_reserved` set. (As we we will see shortly, the name is guaranteed to be there because it is impossible for a client to call `join` without creating a session first, which adds that name to the `_reserved` set.) The code then collects the names of all the users who are currently in the chat room and calls `init` on the callback adapter object to inform the client of the current membership of the room. (We'll discuss why we need this adapter object shortly.)

Having initialized the client, the code now adds the new client's name and callback to the `_members` map and then iterates over the map and invokes the `join` operation on each callback adapter object. This informs all connected clients that a new user has joined the chat room. The notification includes the client that currently executes `join` because we add the new client to the map *before* notifying all the clients. As a result, the client calling `join` may receive events from the chat room while its call to `join` is still executing.

The code for `leave` is very similar, so we do not show it here.

Here is the implementation of `send`:

```
// C++
Ice::Long
ChatRoom::send(const string& name, const string& message)
{
    IceUtil::Mutex::Lock sync(_mutex);
    IceUtil::Int64 timestamp = IceUtil::Time::now().toMilliSeconds();

    MessageEventPtr e = new MessageEvent(timestamp, name, message);
    for(ChatRoomCallbackMap::iterator q = _members.begin(); q != _members.end(); ++q)
    {
        q->second->send(e);
    }
    return timestamp;
}
```

This looks very similar to `join`. The main difference is that, instead of calling `join` on the callback adapter object, the code calls `send`, passing a `MessageEvent` instead of a `UserJoinedEvent`.

The ***`ChatRoomCallbackAdapter`*** *Class*

So, what is the purpose of the `ChatRoomCallbackAdapter` class? When you look back at the code in the `ChatRoom` class, you will see that the class is concerned with passing events to the appropriate clients. In other words, the job of the `ChatRoom` class is to worry about what happens in the chat room, and to whom to send the corresponding events. However, the `ChatRoom` class does not concern itself with *how* to send events to clients. In particular, the `ChatRoom` class does not understand the difference between the push and pull model. Instead of containing explicit code for each model, the `ChatRoom` class delegates the work of making events available to clients to the `ChatRoomCallbackAdapter` class. This class is an abstract base class that we implement once for each of the push and pull models. The concrete implementations of the class then take the appropriate action for each model. Here is the definition of `ChatRoomCallbackAdapter`:

```
// C++
class ChatRoomCallbackAdapter : public IceUtil::Shared
{
public:
    virtual void init(const Ice::StringSeq&) = 0;
    virtual void join(const UserJoinedEventPtr&) = 0;
    virtual void leave(const UserLeftEventPtr&) = 0;
    virtual void send(const MessageEventPtr&) = 0;
};
```

Note that the class simply contains a pure virtual function for each of the operations in the `ChatRoomCallback` interface.

Callback Adapter for the Push Model

For the push model, here is how we implement the adapter class:

```
// C++
class SessionCallbackAdapter : public ChatRoomCallbackAdapter
{
public:
    SessionCallbackAdapter(const ChatRoomCallbackPrx& callback, const ChatSessionPrx& session)
```

```
        : _callback(callback), _session(session)
    {
    }

    void init(const Ice::StringSeq& users)
    {
        _callback->init_async(new AMICallback<AMI_ChatRoomCallback_init>(_session), users);
    }

    void join(const UserJoinedEventPtr& e)
    {
        _callback->join_async(new AMICallback<AMI_ChatRoomCallback_join>(_session),
                              e->timestamp,
                              e->name);
    }

    void leave(const UserLeftEventPtr& e)
    {
        _callback->leave_async(new AMICallback<AMI_ChatRoomCallback_leave>(_session),
                               e->timestamp,
                               e->name);
    }

    void send(const MessageEventPtr& e)
    {
        _callback->send_async(new AMICallback<AMI_ChatRoomCallback_send>(_session),
                              e->timestamp,
                              e->name,
                              e->message);
    }

private:
    const ChatRoomCallbackPrx _callback;
    const ChatSessionPrx _session;
};
```

The constructor remembers the callback proxy of the client's callback object and a session proxy in the `_callback` and `_session` member variables. (We will discuss sessions in the following section.) Each member function simply invokes the corresponding Slice callback operation on the client's callback proxy.

Note that we invoke all operations asynchronously. This is necessary so the server is protected against misbehaved clients. For example, suppose a client's implementation of the `send` operation does not return promptly and blocks for some time. If the server were to invoke the `send` operation synchronously, it would lose a thread of control for the duration of the call. Worse, if you examine the implementation of `ChatRoom::send`, you will see that we iterate over the `_members` map under protection of a lock (to avoid concurrent updates to that map while we are iterating). If the call to `send` were to block, not only would the server lose a thread of control, but it would also prevent execution of other operations until the call completes.

Of course, we could avoid the locking problem by making a copy of the map. However, the `ChatRoomCallbackAdapter` class avoids the need to do this. By providing member functions that are guaranteed not to block, the class not only avoids an inefficient data copy, but also nicely hides the details of asynchronous invocation from the `ChatRoom` class.

Looking at the four member functions of `ChatRoomCallbackAdapter`, you will notice that all four member functions use instances of the same `AMICallback` class to receive notification of when an asynchronous call completes. Here is the definition of this class:

```
// C++
template<class T> class AMICallback : public T
{
public:
    AMICallback(const ChatSessionPrx& session) : _session(session)
    {
    }
```

```
    virtual void ice_response()
    {
    }

    virtual void ice_exception(const Ice::Exception&)
    {
        try
        {
            _session->destroy(); // Collocated
        }
        catch(const Ice::LocalException&)
        {
        }
    }

private:
    const ChatSessionPrx _session;
};
```

If an asynchronous invocation of the client's callback succeeds, this code does nothing. However, if a client's callback operation raises an exception, the chat server responds by immediately destroying that client's session, that is, by evicting the client from the chat room. The rationale here is that, if an invocation on a client's callback object does not work once, it is unlikely to work again later. For example, the client could have passed a callback proxy that points at a non-existent object, in which case *all* invocations on the callback proxy would fail forever. By cancelling the client's membership, the chat server avoids continuing to do work on behalf of dysfunctional clients.

We will return to the callback adapter class for the pull model once we have discussed session creation.

Session Creation for the Push Model

Let us now look at how we can create sessions. For push clients, we use Glacier2 and so use Glacier2's session creation mechanism. Glacier2 permits applications to customize the session creation mechanism by providing a proxy to a `Glacier2::SessionManager` object. You configure Glacier2 to use your session manager by setting the `Glacier2.SessionManager` property. Apart from a trivial constructor that remembers the chat room, the session manager has a single operation, `create`, that Glacier2 calls to delegate session creation to your application. The `create` operation must return a proxy to the session, of type `Glacier2::Session*`. Here is our implementation of this operation:

```
// C++
Glacier2::SessionPrx
ChatSessionManagerI::create(const string& name,

                            const Glacier2::SessionControlPrx&,
                            const Ice::Current& c)
{
    string vname;
    try
    {
        vname = validateName(name);
        _chatRoom->reserve(vname);
    }
    catch(const string& reason)
    {
       throw CannotCreateSessionException(reason);
    }

    Glacier2::SessionPrx proxy;
    try
    {
        ChatSessionIPtr session = new ChatSessionI(_chatRoom, vname);
        proxy = SessionPrx::uncheckedCast(c.adapter->addWithUUID(session));

        Ice::IdentitySeq ids;
```

```
        ids.push_back(proxy->ice_getIdentity());
        sessionControl->identities()->add(ids);
    }
    catch(const Ice::LocalException&)
    {
        if(proxy)
        {
            proxy->destroy();
        }
        throw CannotCreateSessionException("Internal server error");
    }
    return proxy;
}
```

The code calls `validateName`, which is a simple helper function to check that the passed name contains only acceptable characters and maps the name to upper case, and then calls `reserve` on the chat room to add the name to its `_reserved` set. We catch any message thrown by these operations and translate it to `Glacier2::CannotCreateSessionException` because that exception appears in the exception specification of Glacier2's `create` operation.

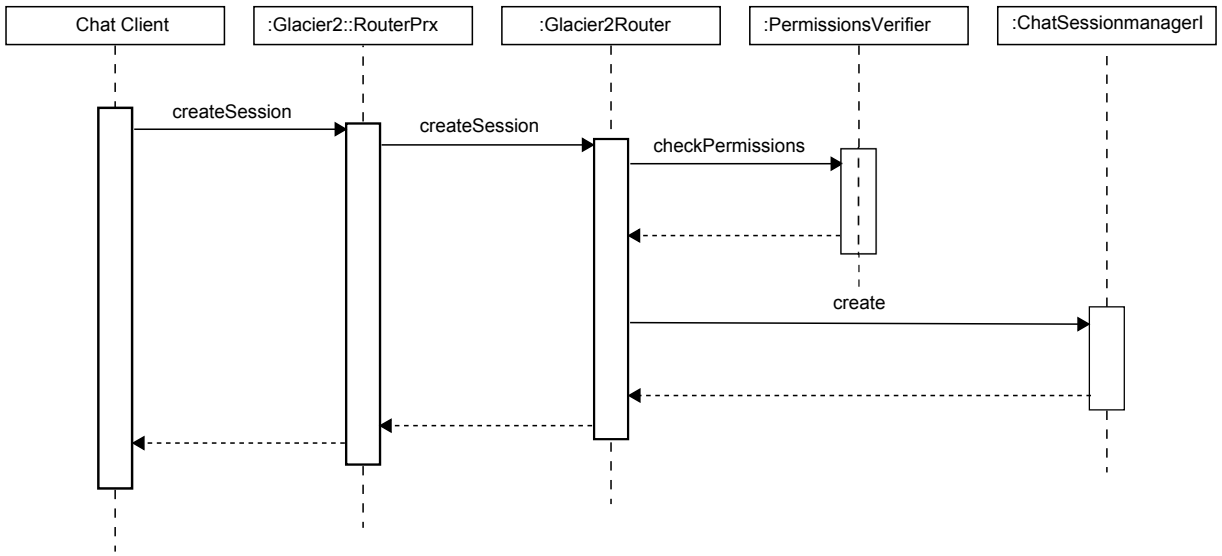
The code then proceeds to create the session by instantiating a `ChatSessionI` object (which we will discuss in a moment). Note that the session uses a UUID as the object identity, so sessions are guaranteed to have unique identifiers.

Finally, the code adds the identity of the newly-created session as the only object to which Glacier2 will forward requests via this session. In effect, this code says "forward requests that arrive via this session only to this session, but no other object." This is a security measure: even if malicious clients can guess the identity of another client's session, this prevents them from sending requests to other objects (possibly in servers other than the chat server) using another client's session. If anything goes wrong, we destroy the just-created session again, to avoid leaking resources.

This is all that is necessary to create a session with Glacier2. If you want to use Glacier2's authentication mechanism, you can do so by setting the property `Glacier2.PermissionsVerifier` to the proxy of an object that performs authentication. (Glacier2 provides built-in permission verifiers, including a `NullPermissionsVerifier` that permits any combination of user name and password.)

Figure 3 shows the sequence of steps that take place during session creation.

Figure 3: Interactions during session creation



Session Implementation for the Push Model

The `ChatSessionI` class implements the `ChatSession` interface:

```
// C++
class ChatSessionI : public ChatSession
{
public:
    ChatSessionI(const ChatRoomPtr&, const string&);

    virtual void setCallback(const ChatRoomCallbackPrx&, const Ice::Current&);
    virtual Ice::Long send(const string&, const Ice::Current&);
    virtual void destroy(const Ice::Current&);

private:
    const ChatRoomPtr _chatRoom;
    const string _name;
    ChatRoomCallbackAdapterPtr _callback;
    bool _destroy;
    IceUtil::Mutex _mutex;
};
typedef IceUtil::Handle<ChatSessionI> ChatSessionIPtr;
```

Apart from the implementations of the four Slice operations, the class has a constructor that remembers the chat room and the client's name. The constructor also sets the `_destroy` member to false.

Let us first consider `setCallback`. When we defined the Slice interface for our session, we glossed over the need for this operation. The operation exists to allow a client, once it has created the session, to pass the proxy to its `ChatRoomCallback` object to the chat server. We must do this with a separate operation because the `Glacier2::create` operation does not allow us to pass this proxy. This means that session creation and setting the callback proxy must be carried out by two separate operations. Here is the implementation of `setCallback`:

```
// C++
void
ChatSessionI::setCallback(const ChatRoomCallbackPrx& callback, const Ice::Current& c)
{
    IceUtil::Mutex::Lock sync(_mutex);
    if(_destroy)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }

    if(_callback || !callback)
    {
        return;
    }

    Ice::Context ctx;
    ctx["_fwd"] = "o";
    _callback = new SessionCallbackAdapter(callback->ice_context(ctx),
                                           ChatSessionPrx::uncheckedCast(
                                               c.adapter->createProxy(c.id)));
    _chatRoom->join(_name, _callback);
}
```

The code first checks whether the session has been destroyed previously and, if so, throws an `ObjectNotExistException`. (See "The Samsara of Objects: Object Life Cycle Operations" in Issue 14 of *Connections* for an explanation of this pattern.) The next step is to check whether the callback was set previously or if the caller passed a null proxy; if so, the operation does nothing. Otherwise, the code creates a new callback adapter object (which we saw previously) and calls `join` on the chat room, which adds the client's name and callback adapter to the `_members` map and notifies all clients that a new user has joined the room.

Also note that the code adds a `_fwd` context with value `"o"` to the client's proxy before passing the proxy to `join`. This instructs Glacier2 to forward callbacks to clients as oneway invocations. Doing so is more efficient than using twoway invocations and, because all callback operations have `void` return type, we might as well use oneway invocations. (See the Ice Manual for more information of the use of contexts with Glacier2.)

The `uncheckedCast` in the preceding code is an idiom to create a proxy to the Ice object that is currently executing: `c.id` is the object identity in the `Current` object that is passed to the operation, so the `uncheckedCast` returns a proxy that contains the executing object's own identity (in other words, a proxy to "self").

The callbacks made by the chat server are ordinary twoway invocations. We use twoway invocations for these callbacks so the chat server is notified about any errors. In turn, this is useful to terminate a client's session when something goes wrong (as we previously discussed in the implementation of the `AMICallback` class).

Once the client has called `setCallback`, the client receives notifications of activity in the chat room. Here is the implementation of the `send` operation:

```
// C++
Ice::Long
ChatSessionI::send(const string& message, const Ice::Current&)
{
    IceUtil::Mutex::Lock sync(_mutex);
    if(_destroy)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
    if(!_callback)
    {
        throw InvalidMessageException("You cannot send messages until you joined the chat.");
    }
    string;
    try
    {
        msg = validateMessage(message);
    }
    catch(const string& reason)
    {
        throw InvalidMessageException(reason);
    }
    return _chatRoom->send(_name, msg);
}
```

As you can see, the operation simply delegates the work of sending the message to the `ChatRoom` class (which, in turn, uses the `ChatRoomCallbackAdapter` class to notify clients). If the client attempts to send a message before it has established its callback object, we throw an `InvalidMessageException`.

The `validateMessage` function is used to limit the length of messages that clients can send to something reasonable. This prevents malicious clients from flooding other clients with traffic. (You can arbitrarily extend message validation to suit your needs. For example, you may want to limit not just size, but also the rate at which clients can send messages.)

To leave the chat room, the client calls `destroy` on the session:

```
// C++
void
ChatSessionI::destroy(const Ice::Current& c)
{
    IceUtil::Mutex::Lock sync(_mutex);
    if(_destroy)
    {
        throw Ice::ObjectNotExistException(__FILE__, __LINE__);
    }
    try
    {
        c.adapter->remove(c.id);
```

```
        if(_callback == 0)
        {
            _chatRoom->unreserve(_name);
        }
        else
        {
            _chatRoom->leave(_name);
        }
    }
    catch(const Ice::ObjectAdapterDeactivatedException&)
    {
        // No need to clean up, the server is shutting down.
    }
    _destroy = true;
}
```

This completes the code for the push model.

Session Creation for the Pull Model

For pull clients, we need to implement the `PollingChatSessionFactory` we defined earlier. Like the Glacier2 session manager, the factory has a single `create` operation, implemented as follows:

```
// C++
PollingChatSessionPrx
PollingChatSessionFactoryI::create(const string& name,
                                   const string& password,
                                   const Ice::Current& c)
{
    string vname;
    try
    {
        vname = validateName(name);
        _chatRoom->reserve(vname);
    }
    catch(const string& reason)
    {
       throw CannotCreateSessionException(reason);
    }

    PollingChatSessionPrx proxy;
    try
    {
        PollingChatSessionIPtr session = new PollingChatSessionI(_chatRoom, vname);
        proxy = PollingChatSessionPrx::uncheckedCast(c.adapter->addWithUUID(session));
        _reaper->add(proxy, session);
    }
    catch(const ObjectAdapterDeactivatedException&)
    {
        throw CannotCreateSessionException("internal server error");
    }
    return proxy;
}
```

The implementation is similar to `create` for push clients: it validates and reserves the user name. (We do not use the password parameter in this case because we do not implement authentication. Of course, if we were to authenticate clients, the relevant check would be implemented here.) In addition, `create` creates a polling chat session, of type `PollingChatSessionI`. The implementation of this class is almost identical to `ChatSessionI`. The main difference is that it also provides an implementation of the `getInitialUsers` operation, which returns the current list of users to the client, and that it uses a different callback adapter implementation, of type `PollCallbackAdapter`. Note that we also add the session to a reaper. We will discuss the reason for this in a moment.

This `PollCallbackAdapter` class is responsible for buffering the current list of users and chat room events until the session calls `getInitialUsers` or `getUpdates`:

```
// C++
class PollCallbackAdapter : public ChatRoomCallbackAdapter
{
public:
    virtual void
    init(const Ice::StringSeq& users)
    {
        IceUtil::Mutex::Lock sync(_mutex);
        _users = users;
    }

    virtual void
    send(const MessageEventPtr& e)
    {
        IceUtil::Mutex::Lock sync(_mutex);
        _updates.push_back(e);
    }

    virtual void
    join(const UserJoinedEventPtr& e)
    {
        IceUtil::Mutex::Lock sync(_mutex);
        _updates.push_back(e);
    }

    virtual void
    leave(const UserLeftEventPtr& e)
    {
        IceUtil::Mutex::Lock sync(_mutex);
        _updates.push_back(e);
    }

    Ice::StringSeq
    getInitialUsers()
    {
        IceUtil::Mutex::Lock sync(_mutex);
        Ice::StringSeq users;
        users.swap(_users);
        return users;
    }

    ChatRoomEventSeq
    getUpdates()
    {
        IceUtil::Mutex::Lock sync(_mutex);
        ChatRoomEventSeq updates;
        updates.swap(_updates);
        return updates;
    }

private:
    Ice::StringSeq _users;
    ChatRoomEventSeq _updates;
    IceUtil::Mutex _mutex;
};
```

The implementation of this class is trivial: it simply keeps a sequence of updates in the private member variable `_updates`. Whenever any activity occurs in the chat room, the `ChatRoom` class calls the corresponding member function on the adapter, which simply appends another event to the `_updates` sequence. Eventually, when the client calls `getUpdates`, that sequence is returned to the client and replaced by a new empty sequence that buffers the next batch of events.

Reaping

Our design requires clients to create a session and, when they leave the chat room, to destroy that session again. This immediately raises the question of "what happens if a client never calls `destroy`?" This could happen, for example, because a client crashes, loses connectivity, or maliciously never destroys its session in an attempt to exhaust resources in the server.

For push clients, this is not a problem: Glacier2 provides a session timeout that we can control via the `Glacier2.SessionTimeout` property. If a client's session remains inactive for longer than this timeout, Glacier2 calls `destroy` on our session, so the server can reclaim resources. (If the machine Glacier2 runs on crashes or the server loses connectivity with Glacier2, that is also not a problem because the chat server will get an error the next time it tries to send an update to a push client and destroy the session in the `AMICallback` class.) However, for pull clients, we must provide our own solution.

The issue of how to clean up resources that are allocated by a server on behalf of clients is a recurring theme in distributed computing. It arises whenever client–server interactions are stateful and your application supports object life cycle. For a general discussion of this topic, see the article "The Grim Reaper" in Issue 3 of *Connections*. For our chat server, we need to make sure that we reclaim the resources associated with a pull session if the client never calls `destroy`:

```
// C++
class ReaperTask : public IceUtil::TimerTask
{
public:
    ReaperTask(int timeout);
    virtual void runTimerTask();
    void add(const PollingChatSessionPrx&, const PollingChatSessionIPtr&);

private:
    IceUtil::Mutex _mutex;
    const IceUtil::Time _timeout;
    list<pair<PollingChatSessionPrx, PollingChatSessionIPtr> > _reapables;
};
typedef IceUtil::Handle<ReaperTask> ReaperTaskPtr;
```

Our `ReaperTask` class stores a timeout and a list of session proxy–servant pairs. The `_timeout` member is the amount of time we will keep an idle pull session alive. The `_reapables` list contains all the pull sessions we want to reap. We store both the proxy and the servant smart pointer so we can invoke both Slice operations and non-Slice member functions.

Instead of implementing a separate thread to reap sessions, we derive our reaper from the `IceUtil::TimerTask` class which does exactly what we need, namely, run a task at specified intervals in a separate thread. The implementation of the constructor and the `add` function are trivial:

```
// C++
ReaperTask::ReaperTask(int timeout) :
    _timeout(IceUtil::Time::seconds(timeout))
{
}

void
ReaperTask::add(const PollingChatSessionPrx& proxy, const PollingChatSessionIPtr& session)
{
    IceUtil::Mutex::Lock sync(_mutex);
    _reapables.push_back(make_pair(proxy, session));
}
```

The actual reaping happens in `runTimerTask`:

```
// C++
void
ReaperTask::runTimerTask()
{
    IceUtil::Mutex::Lock sync(_mutex);
```

```
    list<pair<PollingChatSessionPrx, PollingChatSessionIPtr> >::iterator p =
        _reapables.begin();
    while(p != _reapables.end())
    {
        try
        {
            if((IceUtil::Time::now(IceUtil::Time::Monotonic) - (*p).second->timestamp()) >
                _timeout)
            {
                (*p).first->destroy();
                p = _reapables.erase(p);
            }
            else
            {
                ++p;
            }
        }
        catch(const Ice::LocalException&)
        {
            p = _reapables.erase(p);
        }
    }
}
```

The code simply iterates over the `_reapables` list and calls `destroy` on any session that has not had any activity for the specified timeout. Running the reaper ensures that the server reclaims resources even if pull clients crash, lose connectivity, or neglect to call `destroy`.

*The Server **`main`** Function*

Putting it all together, and using the `Ice::Service` helper class for our server, we end up with the following code:

```
// C++
bool
ChatServer::start(int argc, char* argv[])
{
    _timer = new IceUtil::Timer();
    int timeout = communicator()->getProperties()->getPropertyAsIntWithDefault("ReaperTimeout",
                                                                               10));
    ReaperTaskPtr reaper = new ReaperTask(timeout);
    IceUtil::TimePtr _timer = new IceUtil::Timer();
    _timer->scheduleRepeated(reaper, IceUtil::Time::seconds(timeout));

    try
    {
        Ice::ObjectAdapterPtr adapter = communicator()->createObjectAdapter("ChatServer");

        ChatRoomPtr chatRoom = new ChatRoom();
        adapter->add(new ChatSessionManagerI(chatRoom), communicator()->stringToIdentity(
                "ChatSessionManager"));

        adapter->add(
          new PollingChatSessionFactoryI(chatRoom, reaper),
                  communicator()->stringToIdentity("PollingChatSessionFactory"));

        adapter->activate();
    }
    catch(const Ice::LocalException&)
    {
        _timer->destroy();
        throw;
    }

    return true;
```

```
}

bool
ChatServer::stop()
{
    _timer->destroy();
    return true;
}
```

The code instantiates a reaper with a configurable timeout for pull clients, and then schedules that reaper for repeated execution with an `IceUtil::Timer`. The code then creates an object adapter, and instantiates the chat room. The remainder of the code instantiates the session manager for Glacier2 sessions and the session factory for polling clients, and activates the object adapter. When the service shuts down, the code destroys the timer before it exits.

Implementing a Simple C++ Command-Line Client

Having created our server, we can now turn to the implementation of our first client. This C++ client will be very simple and not bother with asynchronous invocations to avoid blocking. The idea is to provide a simple demonstration of how to use the basic functionality of our application. (We will look at more sophisticated implementation techniques when we present the other clients.) The C++ client will use the push model and connect to the chat server via a Glacier2 session. (Note that the C++ code we ship with the source code for this article also includes a client for the pull model. We provide this client for illustration and to use it in testing, even though, according to our application requirements, a C++ pull client is not needed.)

Implementing the Callback Object

The first step for our client is to implement the `ChatRoomCallback` object that displays activity in the chat room. The implementation of the `send` operation could hardly be simpler:

```
// C++
virtual void
send(Ice::Long, const string& name,
     const string& message, const Ice::Current&)
{
    cout << name << " > " << unstripHtml(message) << endl;
}
```

Whenever a message arrives from the chat server, the client simply prints that message on the terminal. The `unstripHtml` function is a helper function that expands HTML escape sequences, such as `&`.

The implementation of the `join` and `leave` operations prints messages to inform the user that another user has joined or left the room. The implementation is similarly trivial, so we do not show it here.

Creating the Glacier2 Session

To be able to participate in the chat room, our client must create a Glacier2 session, which requires the client to obtain a proxy to the Glacier2 router:

```
// C++
Ice::RouterPrx defaultRouter = communicator()->getDefaultRouter();
if(!defaultRouter)
{
    throw "no default router set";
}

router = Glacier2::RouterPrx::checkedCast(defaultRouter);
if(!router)
{
    throw "configured router is not a Glacier2 router";
}
```

The `getDefaultRouter` operation on the communicator returns a proxy to the client's configured default router. For this to work, the client's configuration must include this proxy, which you can set via the `Ice.Default.Router` property. For example:

```
Ice.Default.Router=Glacier2/router:ssl -p 4064 -h glacier2host
```

The endpoint you specify for this proxy must match Glacier2's setting of the `Glacier2.Clients.Endpoints` property, for example:

```
Glacier2.Client.Endpoints=ssl -p 4064
```

Once the client has obtained the proxy to the Glacier2 router, it is ready to create the session:

```
// C++
try
{
    session = ChatSessionPrx::uncheckedCast(router->createSession(id, pw));
}
catch(const PermissionDeniedException& ex)
{
    cout << "Login failed:\n" << ex.reason << endl;
}
catch(const CannotCreateSessionException& ex)
{
    cout << "Login failed:\n" << ex.reason << endl;
}
catch(const Ice::LocalException& ex)
{
    cerr << "Communication with the server failed:" << ex << endl;
}
```

Assuming that the client passes an acceptable user name and password, the call to `createSession` creates the session and returns its proxy to the client. Note that we down-cast the returned session proxy to `ChatSessionPrx`; this is necessary because the formal return type of `createSession` is `Glacier2::Session` but Glacier2 actually returns a proxy to our custom session, of type `ChatSession`. Of course, for this to work, Glacier2 must be configured to use the chat server's session manager with the `Glacier2.SessionManager` property:

```
Glacier2.SessionManager=ChatSessionManager:tcp -h chathost -p 10001
```

This configuration assumes that the chat server runs on the host `chathost`.

Setting the Callback

Once the client has created the session, it must join the chat room by calling `setCallback` on the session, which passes the callback proxy to the chat server. To actually provide a working callback object, the client must create an object adapter and inform Glacier2 that it wants to receive callbacks from the server. To do this, Glacier2 provides a `getCategoryForClient` operation that returns a unique identifier. Glacier2 associates the client's session with this identifier. In turn, this allows Glacier2 to work out to which of its many clients it should forward a particular callback that it receives from the chat server. The client's job in making this callback mechanism work is to set the `category` member of the object identity of the callback object to the identifier returned by Glacier2:

```
// C++
Ice::ObjectAdapterPtr adapter =
    communicator()->createObjectAdapterWithRouter("Chat.Client", router);
adapter->activate();
Ice::Identity callbackReceiverIdent;
callbackReceiverIdent.name = "callbackReceiver";
callbackReceiverIdent.category = router->getCategoryForClient();
ChatRoomCallbackPtr cb = new ChatRoomCallbackI;
session->setCallback(ChatRoomCallbackPrx::uncheckedCast(adapter->add(cb,
callbackReceiverIdent)));
```

The act of calling `setCallback` joins the client to the chat room and causes the chat server to forward events to the client's callback object.

Keeping the Session Alive

Glacier2 implements a session timeout for push clients. If a session remains idle for longer than the configured timeout, Glacier2 automatically destroys the client's session. To keep the session alive, the client must use the session, that is, invoke at least one operation on the session within each timeout interval. However, we want the client's session to remain alive even if there is no activity in the chat room for, say, a few minutes, even if Glacier2's timeout is set to something shorter. We can do this by periodically calling an operation that does nothing on the session. Rather than explicitly defining such an operation, we can conveniently call `ice_ping` to cause activity. A timer task makes it easy to schedule these calls:

```
// C++
class SessionRefreshTask : public IceUtil::TimerTask
{
public:
    SessionRefreshTask(const ChatSessionPrx& session) :
        _session(session),
        _destroyed(false)
    {
    }

    virtual void
    runTimerTask()
    {
        IceUtil::Mutex::Lock sync(_mutex);
        bool destroyed = _destroyed;
        sync.release();

        if(!destroyed)
        {
            try
            {
                _session->ice_ping();
            }
            catch(const Ice::LocalException& ex)
            {
                sync.acquire();
                _destroyed = true;
                if(!dynamic_cast<const ObjectNotExistException*>(&ex))
                {
                    cerr << "session lost: " << ex << endl;
                }
            }
        }
    }

    bool isDestroyed()
    {
        IceUtil::Mutex::Lock sync(_mutex);
        return _destroyed;
    }

private:
    const ChatSessionPrx _session;
    bool _destroyed;
    IceUtil::Mutex _mutex;
};
typedef IceUtil::Handle<SessionRefreshTask> SessionRefreshTaskPtr;
```

This task calls `ice_ping` on the session and, once the session is destroyed, sets a flag (returned by `isDestroyed`). The `isDestroyed` function is useful to control the main loop of the client, which reads messages from the user's keyboard and displays messages that are sent to the chat room on the screen: the `isDestroyed` function can be used to terminate that loop. Note that the code releases the mutex before calling `ice_ping`, in case `ice_ping` blocks

for some time. (This is unlikely but could happen due to network problems.) The code prints an error message for all exceptions other than `ObjectNotExistException`. (The latter is expected—`ObjectNotExistException` is raised once the client has called `destroy` on its session.)

Before entering its main loop (which we do not show here), the client instantiates the timer task to ensure that the session does not time out:

```
// C++
Ice::Long refreshPeriod = router->getSessionTimeout() / 2;
SessionRefreshTaskPtr refreshTask = new SessionRefreshTask(session);
IceUtil::TimerPtr timer = new IceUtil::Timer();
timer->scheduleRepeated(refreshTask, IceUtil::Time::seconds(refreshPeriod));
```

Note that we set the refresh period to half the session timeout. This ensures that `ice_ping` will be called at least once during each timeout interval and makes some allowance for network delays.

Why not IceStorm?

Before we finish this article, we should answer a question you may have had in mind all along: "Why not use IceStorm for this application?"

The first (and somewhat glib) answer is that, had we done that, we wouldn't have had a cool demo to write about. The issues we discussed in this article (such as the need for sessions to clean up server-side resources and the need to avoid blocking) arise in many distributed applications, and it is important to have a clear understanding of these issues and their solutions.

The second (and less glib) answer is that "yes, if not for demonstration purposes, we indeed would have used IceStorm" After all, IceStorm is a service that is specifically built to efficiently distribute messages that have one source but many destinations, which is a very good match for our application. Moreover, IceStorm would achieve the distribution of messages more efficiently because of a number of optimizations that are not present in our implementation.

However, IceStorm, while providing a good starting point, is not a complete solution for our application:

- IceStorm can distribute updates to the chat room efficiently to clients, but does not provide any mechanism for clients to get the initial membership of the room.
- IceStorm does not provide a means to limit messages based on how frequently they are sent.
- IceStorm does not provide a means to reject messages beyond a certain size (other than by setting `Ice.MessageSizeMax`).
- It is not clear how clients would select an IceStorm topic for the chat room.
- IceStorm only supports a push model and cannot directly accommodate pull clients.
- IceStorm does not have a built-in session concept.
- IceStorm does not provide an authentication mechanism.

So, by itself, IceStorm cannot meet our requirements. However, for our application, we could use IceStorm to do part of the job cheaply, namely, to distribute updates to push clients. To accommodate pull clients, we could use a technique similar to the one we used for our implementation: have IceStorm push updates to an interposed push client which then buffers these updates and offers a pull interface to the (real) pull clients.

To handle other requirements, such as the need to limit message size and to authenticate clients, we would have to write additional code to provide the functionality that IceStorm does not provide as built-in features. In effect, using IceStorm in this way means that we would treat it as an internal infrastructure service rather than as a complete application. This is an approach that you will frequently encounter during development. Rather than trying to provide a complete solution, Ice services (such as IceStorm) provide core functionality that you would find difficult to implement yourself (at least to the same degree of reliability and performance). The idea is that, by supplementing an Ice service with application-specific functionality, you can build your application quicker than if you had to develop all the functionality yourself, without having to compromise efficiency and scalability.

Summary

This article showed you how to write an industrial-strength Ice application. By necessity, doing this involves more code than you are probably used to seeing in these articles. There are several reasons for this:

- We have set requirements that reflect real-world needs and go beyond a simple toy application.
- Because we want to support PHP and Silverlight bridge clients, we had to implement both a push and a pull model.
- The server has to be robust in the face of misbehaved clients, which necessitates the use of asynchronous invocations and the need to clean up resources in the server if clients crash or forget to destroy their session.
- We have structured the code such that it reflects real-world software engineering practices by cleanly separating implementation concerns, avoiding the use of hard-wired constants for timeouts, and providing real error handling.

As you can see, taking care of real-world concerns does add to the amount of code we need to write. However, keep in mind that much of this code is boiler plate, in the sense that it implements recurring patterns in distributed computing. For example, the need for sessions is mandated by Glacier2 as well as the need to reclaim resources in the server. The reaping implementation to do so is a standard solution to a standard problem. In other words, once you have implemented reaping once, you will recognize the pattern and re-apply it to other applications with little effort. (And, with a bit of thought, you can easily create a reusable reaping implementation so, the second time around, using it becomes a zero-cost exercise.)

Similarly, clients need to refresh sessions periodically to prevent them from timing out. Again, this is a standard problem with a standard solution, and a few lines of code, together with `IceUtil::Timer`, are sufficient to deal with the issue.

So, while we have presented more code than usual, do not let this discourage you: any application that must run under real-world conditions will have to deal with similar issues and will require a similar amount of code to deal with them.

In the next article, we will see how we can create the remaining push and pull clients. As always, you can download the complete source code for this article. We encourage you to study the code and to experiment with the live chat server we are running on our web site.