

Let's Chat! (Part 2)

José Gutiérrez de la Concha, Software Developer
Mark Spruiell, Software Developer

Introduction

In [part one](#) of this article series, we covered a great deal of ground by introducing the chat application's requirements, discussing the issues associated with the push and pull models for event delivery, reviewing the Slice definitions, and describing salient aspects of the implementations of the C++ server and the C++ push client. In this second and final article of the series, we conclude with an examination of the graphical applications and the web-based chat clients.

Contents

Introduction	1
Developing a Graphical Client.....	1
Glacier2 Integration	2
Sending a Message.....	3
.NET Chat Client	4
Java Chat Client.....	13
Web Clients	14
Silverlight Clients.....	25
Summary	35

Developing a Graphical Client

Our application includes two functionally equivalent graphical clients written in different languages. The Java client uses the Swing tool kit to build its user interface, and the C# client employs the Windows Presentation Foundation (WPF) of .NET 3.5. As you will see, the programming idioms are the same for both clients, despite the differences in their underlying frameworks.

One of the most important considerations in any graphical program is the responsiveness of the user interface. Many modern GUI tool kits use a single-threaded design in an effort to simplify the common use cases, with the limitation that updates to the user interface can only be performed by a special thread that we'll call the *event loop thread*. While executing in the context of this thread, for example in response to a user interface event such as a button press, delays of any significant duration can negatively impact the user experience. Whether a program performs a complex image transformation or issues a series of remote invocations, maintaining a responsive user interface is generally a top priority.

When building a graphical program with Ice, you might be tempted to issue a synchronous remote invocation from the event loop thread, optimistically assuming that the invocation won't block. Unfortunately, a remote invocation can block due to circumstances beyond the control of the Ice run time, such as sluggish DNS lookups, congested networks, or unresponsive servers. Our recommendation is therefore quite simple: *never* make a synchronous invocation from the event loop thread. Instead, you should use asynchronous invocations, which (as of Ice 3.3) are guaranteed not to block.

Changing an application to use Ice's asynchronous method invocation (AMI) model involves the following tasks:

- Creating appropriate callbacks for each operation that the program invokes via AMI. The Ice run time invokes the callbacks to notify the application about the success or failure of an operation.
- Calling the asynchronous version of the proxy method instead of the synchronous one.

In addition, you need to keep in mind the threading semantics of AMI callbacks; understanding these is vital for the proper operation of a graphical program. A client will typically need to update the user interface after being notified about the completion of an AMI request, but frameworks such as Swing and WPF do not permit updates to the user interface from any thread except the event loop thread. Since the Ice run time invokes AMI callbacks from its own threads by default, it is not safe for a callback to directly manipulate the user interface.

Ice provides an easy way to deal with this issue. A graphical program can install its own implementation of the `Ice.Dispatcher` facility to force Ice callbacks to execute in the event loop thread and therefore make it safe for callbacks to update the GUI directly. The graphical clients presented in this article employ this facility to simplify the use of asynchronous invocations.

Our graphical clients both use the push model, meaning they receive callbacks from the chat server (via a Glacier2 router) to notify them about changes in the chat room, such as the addition of another user or the publication of a new chat message. As with every push client, our programs implement the `ChatRoomCallback` interface to receive these notifications. We will begin our review with the C# client, but the Java client is identical in many respects.

The remainder of this article assumes that you are familiar with Ice's asynchronous programming model. For more information on using AMI in Ice, please refer to the relevant client-side language mapping chapters of the [Ice manual](#).

Note that the deprecated `["ami"]` metadata tag shown in our Slice definitions is only necessary for Silverlight clients. The Java and .NET clients use the new AMI mapping introduced in Ice 3.4, which doesn't require this tag.

Glacier2 Integration

Programs that interact with a Glacier2 router need to handle several administrative tasks unrelated to the application domain:

- Configure Ice to use the router

- Create a session
- Keep the session alive
- Create an object adapter for receiving callbacks from the router
- Create and register callback objects

To simplify the development of GUI applications that interact with Glacier2, Ice provides helper classes that manage most of these tasks for you:

- `Glacier2.SessionFactoryHelper`

This class simplifies the creation of Glacier2 sessions. It allows you to configure all of the settings necessary for establishing a connection to the router, including its host name, port number, object identity, timeout, and whether to use an SSL connection. (The class also provides default values for these settings.) Once configured, the factory can be used to create `SessionHelper` objects.

- `Glacier2.SessionHelper`

This class provides functionality similar to the `Glacier2::Application` class that we discussed in the first article. It initializes the Ice run time, keeps a Glacier2 session alive, eases the creation of callback objects, and notifies the application about important events in the session's lifecycle.

- `Glacier2.SessionCallback`

An application can implement this interface to receive notification about session lifecycle events.

The Java and C# push clients presented in this article make use of these helper classes.

Sending a Message

To demonstrate the use of asynchronous invocations in Java and C#, let's use the `send` operation as an example. Recall the Slice definition of `send` from part one:

```
// Slice
["ami"] long send(string message) throws InvalidMessageException;
```

The operation accepts a message string and returns a long integer representing a time stamp. If the server deems the message unacceptable for any reason, the operation raises `InvalidMessageException`.

A client uses this operation to publish a new chat message, which prompts the server to send an update to every chat client, including the one that originated the message. This presents us with an implementation decision: When do we add the message that the user just typed to the chat history in the user interface? There are several possibilities:

- Add the message immediately, just before or after the asynchronous call to `send`.

- Don't add the message until we have confirmed that the call to `send` has completed successfully.
- Handle the chat message just like a message from any other user by waiting until the client receives its chat update from the server.

The first option provides instant visual confirmation that the program has acted on the user's command. However, in reality, this is not a viable choice because our chat history includes the time stamp that the server assigns to each message, but we do not know the time stamp of the just-sent message until the asynchronous invocation of `send` completes.

If we use the second option, we need to address the issue of preventing duplicate entries in the chat history when the client eventually receives its chat update from the server. In other words, if we have already added the user's message to the chat history, we do not want to add it a second time when the server publishes the message to every client.

The last option is the simplest: take no action when sending the message, and instead wait for the update from the server before adding the message to the chat history.

This decision is more important for the pull clients than for the push clients. Because the pull clients must poll for chat updates at regular intervals, the user interface appears more responsive if the message is displayed immediately after `send` completes rather than waiting until the next polling interval. As far as the graphical clients are concerned, either solution would be acceptable, but we opted for the second one primarily for the sake of consistency with the pull clients.

.NET Chat Client

Implementing a Dispatcher

Configuring the Ice run time with a dispatcher ensures that callback replies are invoked from the event loop thread. The dispatcher must be specified during communicator initialization, therefore we create an `Ice.InitializationData` object and set its `dispatcher` member as shown below:

```
// C#
Ice.InitializationData initData = new Ice.InitializationData();
// ...
initData.dispatcher =
    delegate(System.Action action, Ice.Connection connection)
    {
        if(!_exit) // The GUI is being destroyed,
        {          // don't use the GUI thread any more
            action();
        }
        else
        {
            System.Windows.Application.Current.Dispatcher.BeginInvoke(
                DispatcherPriority.Normal, action);
        }
    };
//...
```

Our custom dispatcher implementation checks the `_exit` member, which is set to true when the program is in the process of shutting down. In this case the dispatcher executes the given action

directly, otherwise it hands the action off to `Dispatcher.BeginInvoke`, which is WPF's mechanism for asynchronously executing code within the context of the event loop thread. This `Ice.InitializationData` object is eventually used to initialize a communicator, as we'll see later.

Using AMI

In order to call `send` asynchronously, we first need to define the operations to handle the response and exception conditions of the operation; however, our design has the additional requirement to keep the message around so that we can display it as soon as the invocation completes.

```
// C#
public class AMI_ChatSession_sendI
{
    public AMI_ChatSession_sendI(Coordinator coordinator, string name,
                                string message)
    {
        _coordinator = coordinator;
        _name = name;
        _message = message;
    }

    public void response(long timestamp) { ... }
    public void exception(Ice.Exception ex) { ... }

    private Coordinator _coordinator = null;
    private string _name = "";
    private string _message = "";
}
```

The constructor for this class takes several arguments that it saves as member data for later use in the callback methods: the `Coordinator` object encapsulates most of the client logic, `name` represents the name of the current user, and `message` is the chat message that is in the process of being published to the chat room. We need all of these values to implement the strategy we discussed earlier, namely adding the message to the chat history in the user interface as soon as the call to `send` completes. In `response` we see how these values are used:

```
// C#
public void response(long timestamp)
{
    _coordinator.userSayEvent(timestamp, _name, _message);
}
```

The implementation invokes a method on the coordinator and passes the results of the `send` operation (the time stamp) along with the name and message that we saved in the constructor. As you will see later, the client contains additional logic to prevent duplicate messages when processing chat updates from the server.

Our client must also handle errors in a reasonable way; we decided to display errors in the chat history rather than in a dialog box. The implementation of the `exception` method is shown below:

```
// C#
public void exception(Ice.Exception ex)
{
    if(ex is Chat.InvalidMessageException)
```

```

    {
        Chat.InvalidMessageException e = (Chat.InvalidMessageException)ex;
        _coordinator.appendMessage("<system-message> - " + e.reason +
                                   Environment.NewLine);
    }
    else
    {
        _coordinator.destroySession();
    }
}

```

If `send` raises `InvalidMessageException`, we report a warning but allow the user to remain in the current session. If any other exception occurs, we consider that a fatal error condition and immediately terminate the user's session.

(Keep in mind that `response` and `exception` are called from our custom dispatcher, so it is safe to update the GUI directly.)

Now that our AMI callback is defined, the task of invoking `send` asynchronously is trivial:

```

// C#
ChatSessionPrx _chat = ...;
AMI_ChatSession_sendI cb =
    new AMI_ChatSession_sendI(this, _username, message);
_chat.begin_send(message).whenCompleted(cb.response, cb.exception);

```

Given a proxy to our chat session, we invoke `begin_send` and pass the message to send as the only argument. The return value of `begin_send` is an `Ice.AsyncResult` object; we invoke `whenCompleted` on this object to associate our callback delegates with the invocation.

Receiving Chat Updates

As push clients, our graphical programs need to implement the `ChatRoomCallback` interface, which is defined as follows:

```

// Slice
interface ChatRoomCallback
{
    void init(Ice::StringSeq users);
    void send(long timestamp, string name, string message);
    void join(long timestamp, string name);
    void leave(long timestamp, string name);
};

```

The C# implementation of `ChatRoomCallback` is trivial because it forwards all of the invocations to the `Coordinator` object, which performs the actual work of updating the user interface. However, the implementation of `send` is worth noting:

```

// C#
public class ChatRoomCallbackI : Chat.ChatRoomCallbackDisp_
{
    public ChatRoomCallbackI(Coordinator coordinator)
    {
        _coordinator = coordinator;
    }
}

```

```

public override void send(long timestamp, string name, string message,
                          Ice.Current current)
{
    if(!name.Equals(_coordinator.getUsername(),
                    StringComparison.CurrentCultureIgnoreCase))
    {
        _coordinator.userSayEvent(timestamp, name, message);
    }
}
// ...
}

```

We mentioned earlier that we would need to take measures to prevent duplicate chat messages, and here you can see our solution: ignore any messages that originated from the current user. Since the code that invokes `ChatSession::send` also handles adding the message to the chat history, no action is necessary when that message returns to the client in the form of a chat update.

Glacier2 Integration

Our client application uses the `Glacier2.SessionFactoryHelper` and `Glacier2.SessionHelper` classes to simplify the task of integrating with Glacier2. Furthermore, our `Coordinator` class implements the `Glacier2.SessionCallback` interface so that it can respond to session lifecycle events.

Implementing the `SessionCallback` Interface

The `Glacier2.SessionCallback` interface notifies the application about several events in the lifecycle of a session:

```

// C#
namespace Glacier2
{
    public interface SessionCallback
    {
        void createdCommunicator(SessionHelper session);
        void connected(SessionHelper session);
        void disconnected(SessionHelper session);
        void connectFailed(SessionHelper session, Exception ex);
    }
}

```

An application can optionally supply an object that implements this interface when using the Glacier2 helper classes. In turn, the helper classes ensure that the callback methods are invoked using the same dispatching semantics as the Ice run time, therefore it is safe to update the GUI directly from these callbacks when a custom dispatcher is installed.

Let's examine how the `Coordinator` class implements these methods, beginning with `connectFailed`:

```

// C#
public void connectFailed(Glacier2.SessionHelper session,
                        Exception exception)
{
    //
    // Ignore callbacks during shutdown.
    //
}

```

```

    if(_exit)
    {
        return;
    }

    try
    {
        throw exception;
    }
    catch(Glacier2.CannotCreateSessionException ex)
    {
        setError("Login failed (Glacier2.CannotCreateSessionException):\n"
            + ex.reason);
    }
    catch(Glacier2.PermissionDeniedException ex)
    {
        setError("Login failed (Glacier2.PermissionDeniedException):\n" +
            ex.reason);
    }
    // ...
    catch(System.Exception ex)
    {
        setError("Login failed:\n" + ex.ToString());
    }
}

```

The `connectFailed` method is called if an exception occurs during session creation. The method has two arguments: a reference to the session helper object that encountered the exception, and the exception itself.

Our implementation first examines the `_exit` member and returns immediately if it's true. (The coordinator sets this flag when the program is in the process of shutting down.) Next we re-throw the exception so that we can tailor the error message that we pass to `setError`. The `setError` method decides how to present the message depending on the application's current state: if the application is disconnected, the message is displayed in a popup window, otherwise if the client is already connected the message is displayed in the chat history view.

The `createdCommunicator` method notifies the application that a communicator has been initialized. This method is called prior to session creation (or any other remote invocation), providing the application with an opportunity to tailor the communicator if necessary. We use this hook to install a custom certificate verifier in the IceSSL plug-in:

```

// C#
public void createdCommunicator(Glacier2.SessionHelper session)
{
    //
    // Ignore callbacks during shutdown.
    //
    if(_exit)
    {
        return;
    }

    Ice.Communicator communicator = session.communicator();
    if(communicator.getProperties().getProperty(
        "IceSSL.ImportCert.CurrentUser.Root").Length == 0)

```

```

{
    // Read the CA certificate bundled with this executable and add
    // it to the CurrentUser Root store.
    X509Certificate2 caCert = null;
    X509Store store =
        new X509Store("Root", StoreLocation.CurrentUser);
    try
    {
        store.Open(OpenFlags.ReadWrite);
        System.Reflection.Assembly assembly =
            System.Reflection.Assembly.GetExecutingAssembly();
        System.IO.Stream stream = assembly.GetManifestResourceStream(
            "ChatDemoGUI.zeroc_ca_cert.pem");
        if(stream != null)
        {
            byte[] buf = new byte[stream.Length];
            stream.Read(buf, 0, buf.Length);
            stream.Close();
            caCert = new X509Certificate2(buf);
            store.Add(caCert);
        }
    }
    finally
    {
        store.Close();
    }

    // Install a certificate verifier to ensure the
    // server certificate is signed by the CA cert that
    // is bundled with the executable.
    IceSSL.Plugin plugin = (IceSSL.Plugin)communicator.
        getPluginManager().getPlugin("IceSSL");
    plugin.setCertificateVerifier(new CertificateVerifier(caCert));
}
}

```

Note that the implementation can access the newly-created communicator object by calling the `communicator` method on the given `SessionHelper` argument.

The code tests for the presence of the `IceSSL.ImportCert.CurrentUser.Root` property and, if it's undefined, we assume that the user wants to use the development certificates that are included with the demo. In this case we read the Certificate Authority (CA) certificate from a bundled resource, add it to the certificate store, and then configure the IceSSL plug-in with a certificate verifier. The verifier ensures that the certificate provided by the server is signed by the CA certificate that we just loaded.

The `connected` method in the `SessionCallback` interface notifies the client application that a Glacier2 session has been successfully established:

```

// C#
public void connected(Glacier2.SessionHelper session)
{
    //
    // Ignore callbacks during shutdown.
    //
    if(_exit)
    {
        return;
    }
}

```

```

    }

    // If the session has been reassigned avoid the spurious callback.
    if(session != _session)
    {
        return;
    }

    Chat.ChatRoomCallbackPrx callback =
        Chat.ChatRoomCallbackPrxHelper.uncheckedCast (
            _session.addWithUUID(new ChatRoomCallbackI(this)));

    _chat = Chat.ChatSessionPrxHelper.uncheckedCast (_session.session());
    try
    {
        _chat.begin_setCallback(callback).whenCompleted(
            delegate()
            {
                _username = ChatUtils.formatUsername(_info.Username);
                _info.save();
                setState(ClientState.Connected);
            },
            delegate(Ice.Exception ex)
            {
                destroySession();
            });
    }
    catch(Ice.CommunicatorDestroyedException)
    {
        // Ignore - client session was destroyed.
    }
}

```

The code first checks that the session helper from which it received the callback is still the session helper being used by the application. If the session doesn't match, the method returns immediately to ignore the spurious callback.

Next, the code creates the client's callback object and a proxy to the callback object, all in a single line of code:

```

Chat.ChatRoomCallbackPrx callback =
    Chat.ChatRoomCallbackPrxHelper.uncheckedCast (
        _session.addWithUUID(new ChatRoomCallbackI(this)));

```

Note that our code hasn't initialized any object adapter. The `SessionHelper` object provides an object adapter especially for use by callback objects; this object adapter is created automatically upon the first call to the helper's `objectAdapter` or `addWithUUID` methods.

The coordinator then saves the `ChatSessionPrx` proxy in its `_chat` member for later use when sending messages to the chat room:

```

_chat = Chat.ChatSessionPrxHelper.uncheckedCast (_session.session());

```

The call to `_session.session()` returns a proxy of type `Glacier2.SessionPrx` that we down-cast to the type `Chat.ChatSessionPrx`. We use `uncheckedCast` here and not `checkedCast` for two reasons: a checked cast internally issues a synchronous remote invocation, which is not

something we want to do from the event loop thread, and furthermore we know that the session object supports the `Chat::ChatSession` interface so there is no need to use a checked cast to confirm this fact.

With the client's session proxy and callback proxy in place, we are ready to connect to the chat room by calling the `setCallback` operation. Since we are executing in the event loop thread, we invoke `setCallback` asynchronously and configure its success and failure delegates appropriately.

The last remaining method of the `SessionCallback` interface, `disconnected`, notifies the application that the session has ended:

```
// C#
public void disconnected(Glacier2.SessionHelper session)
{
    //
    // Ignore callbacks during shutdown.
    //
    if(_exit)
    {
        return;
    }

    _username = "";
    if(_model.State == ClientState.Disconnecting)
        // Connection closed by user logout/exit
    {
        setState(ClientState.Disconnected);
    }
    else if(_model.State == ClientState.Connected)
        // Connection lost while user was chatting
    {
        setError(
            "<system-message> - The connection with the server " +
            "was unexpectedly lost.\n" +
            "You can try Login again from the File menu.");
    }
    else // Connection lost while user was connecting
    {
        setError(
            "<system-message> - The connection with the server " +
            "was unexpectedly lost.\n" +
            "Try again.");
    }
}
```

The code behaves differently depending on the current state of the application. For example, the user may have initiated the destruction of the session by explicitly requesting to log out or exit the client, in which case the invocation of `disconnected` is expected. Otherwise, the code logs an error message and encourages the user to try again.

Creating a `SessionFactoryHelper` Object

The constructor for our `Coordinator` class prepares an `Ice.InitializationData` object and uses it to initialize an instance of `SessionFactoryHelper`:

```
// C#
public Coordinator(string[] args)
```

```

{
    //...
    _args = args;
    Ice.InitializationData initData = new Ice.InitializationData();
    initData.properties = Ice.Util.createProperties(ref _args);
    initData.properties.setProperty("Ice.Plugin.IceSSL",
                                   "IceSSL:IceSSL.PluginFactory");
    initData.properties.setProperty("IceSSL.TrustOnly.Client",
                                   "CN=Glacier2");

    initData.dispatcher =
        delegate(System.Action action, Ice.Connection connection)
        {
            ...
        };

    _factory = new Glacier2.SessionFactoryHelper(initData, this);
}

```

The code creates a property set that includes any properties defined on the command line, and defines some additional properties to configure the IceSSL plug-in. The inclusion of the `IceSSL.TrustOnly.Client` property is a security measure that allows an outgoing SSL connection to be established only if the remote peer supplies a certificate whose common name (CN) matches `Glacier2`. (The client's peer is always a Glacier2 router.) This extra level of validation occurs after .NET has successfully executed its low-level certificate verification phase.

Next, the constructor installs a custom dispatcher that ensures Ice callbacks are dispatched on the event loop thread, as we discussed earlier.

Finally, we instantiate a `SessionFactoryHelper` object and pass it our `InitializationData` and a `SessionCallback`, which in this case is the `Coordinator` object itself. We'll see how the `SessionFactoryHelper` object gets used in the next section.

Establishing a Session

The coordinator's `login` method is called from a GUI event handler to begin the process of establishing a Glacier2 session:

```

// C#
public void login(LoginInfo info)
{
    setState(ClientState.Connecting);
    _info = info;
    _factory.setRouterHost(info.Host);
    _session = _factory.connect(info.Username, info.Password);
}

```

The `LoginInfo` argument encapsulates the user-supplied details necessary for establishing a session, such as the host name on which the router is running and the account name and password. The code configures the `SessionFactoryHelper` object with the router's host name; we leave the remainder of the factory's settings (port number, timeout, and so on) unchanged because its default values are acceptable.

Next `login` invokes the factory's `connect` method. The return value is a reference to a `SessionHelper` object. Note however that, in order to avoid blocking the calling thread, the `connect` method uses asynchronous invocations to create the Glacier2 session. As a result, it is not

safe to assume that the session has been successfully established at this time. Rather, we must wait for the `SessionCallback` methods to be invoked to discover the outcome of the request.

Java Chat Client

Given that the chat client implementations in .NET and Java are so similar, we will not repeat all of the material from the previous section but rather highlight where the Java client differs from its .NET counterpart.

Implementing a Dispatcher

As in the C# client, we need to install a custom implementation of `Ice.Dispatcher` to ensure that Ice callback invocations occur in the event loop thread:

```
// Java
initData.dispatcher = new Ice.Dispatcher()
{
    public void dispatch(Runnable runnable, Ice.Connection connection)
    {
        if(_exit) // The GUI is being destroyed,
        {
            // don't use the GUI thread any more
            runnable.run();
        }
        else
        {
            SwingUtilities.invokeLater(runnable);
        }
    }
};
```

Our Java implementation creates an anonymous class whose `dispatch` method delegates to `Swing's invokeLater` function unless the `_exit` member is true, in which case the `Runnable` is executed immediately.

Using AMI

Our AMI callback for `send` must implement two versions of `exception` because `Ice.UserException` and `Ice.LocalException` do not share a common base class as they do in .NET:

```
// Java
public class AMI_ChatSession_sendI extends Chat.Callback_ChatSession_send
{
    public AMI_ChatSession_sendI(String name, String message)
    {
        _name = name;
        _message = message;
    }

    public void response(long timestamp) { ... }
    public void exception(Ice.UserException ex) { ... }
    public void exception(Ice.LocalException ex) { ... }

    private final String _name;
    private final String _message;
}
```

The constructor also differs from the one for the .NET client: it does not require a reference to the coordinator object because we've defined the callback class as an inner class of `Coordinator`. This means the implementation of `response` can simply invoke a method on its outer `Coordinator` object to add the message to the chat history:

```
// Java
public void response(long timestamp)
{
    userSayEvent(timestamp, _name, _message);
}
```

The only user exception that our callback must handle is `InvalidMessageException`. The exception method adds an error message to the chat history:

```
// Java
public void exception(Ice.UserException ex)
{
    if(ex instanceof Chat.InvalidMessageException)
    {
        Chat.InvalidMessageException e = (Chat.InvalidMessageException)ex;
        _chatView.appendMessage("<system-message> - " + e.reason);
    }
}
```

Since a local exception is considered a fatal error condition, we need to terminate the user's session, reclaim resources, and display a message in the chat history. The `destroySession` method handles these tasks for us:

```
// Java
public void exception(Ice.LocalException ex)
{
    destroySession();
}
```

Web Clients

The web has matured into a very capable platform for many types of client applications. The ability to significantly reduce deployment costs, coupled with the increasingly sophisticated interactive features found in modern browsers, makes the web an attractive environment for hosting both corporate and consumer applications.

Among the various language mappings that Ice supports, two are targeted expressly at web-based clients: Ice for PHP and Ice for Silverlight. Although they share a similar purpose, their use cases are quite different. PHP is a server-side scripting language whose interpreter typically executes as a module within the web server; the Ice extension for PHP therefore enables PHP scripts to invoke operations on any Ice server. A web-based client that employs Ice for PHP normally uses HTML and CSS for its user interface and Ajax (asynchronous JavaScript and XML) to create a more dynamic user experience and communicate with the back-end PHP scripts that perform the application logic.

In contrast, a Silverlight web client executes inside a browser plug-in similar to a Java applet, and Ice for Silverlight allows such a client to issue remote invocations directly from the browser. Currently,

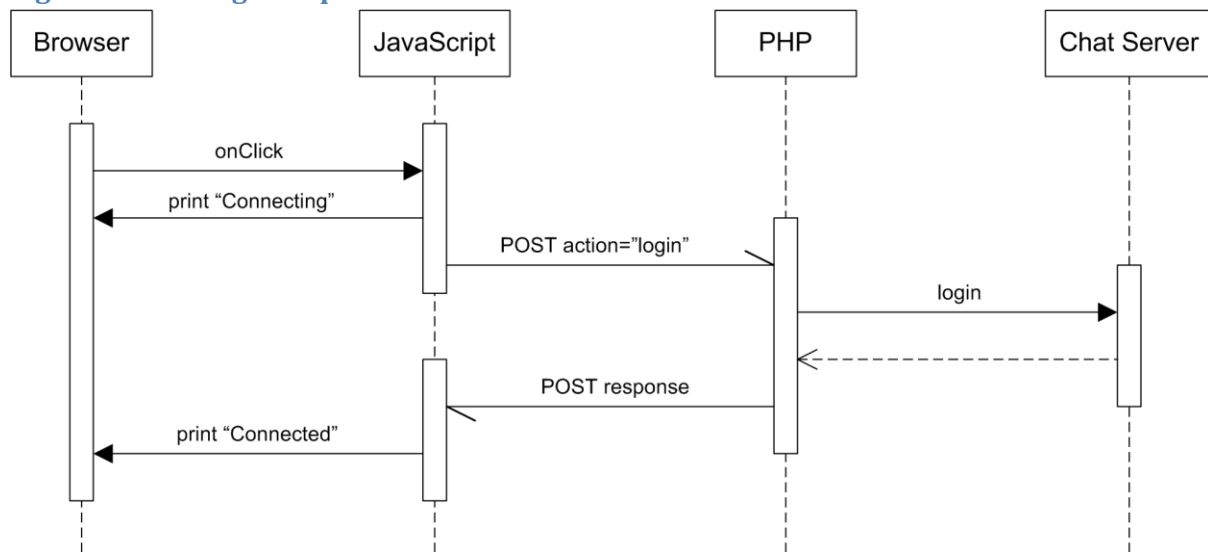
Ice for Silverlight supports two different transports: Ice requests over regular sockets forwarded by a Glacier2 router, and HTTP (or HTTPS) `POST` requests forwarded by an ASP.NET bridge.

The chat demo includes a PHP client that uses the pull model, a Java applet client that uses the push model (and has many similarities with the GUI clients we explored earlier in this article), and two Silverlight clients: one that uses the socket transport and the push model, and another that uses the pull model over the HTTPS transport with the ASP.NET bridge. We'll examine each of these clients in depth.

PHP Client

The PHP chat "client" consists of two components: the HTML and JavaScript front end presented in the browser, and the PHP scripts running in the web server. Ice is not involved in the communication between the front end and the PHP scripts; instead, the JavaScript code issues `POST` requests over HTTPS that prompt the PHP scripts to invoke Ice operations on the back-end chat server. The PHP scripts encode their results using the JavaScript Object Notation (JSON), a lightweight data interchange format, which is decoded in JavaScript and subsequently used to update the user interface. Figure 1 shows how the various components interact when the user clicks the `Login` button.

Figure 1: PHP Login Sequence



Using asynchronous JavaScript instead of traditional HTML forms to issue `POST` requests minimizes bandwidth consumption and, more importantly, gives a browser-based application a look and feel that aligns the user experience more closely to that of a desktop application. For example, upon pressing the `Send` button to submit a new chat message, a simple form implementation would cause the entire user interface to be repainted. We can avoid such distracting interruptions (at the expense of some additional complexity) by using snippets of JavaScript that execute in the background to update individual components of the user interface as necessary.

The PHP Middle Tier

To fully understand the PHP scripts that comprise the middle tier of our client, you must first understand some fundamental concepts of programming in PHP. The interpreter uses an architecture in which each invocation of a PHP script (called a request) is stateless and therefore

independent of any other request. Essentially, each request operates as if it were in its own private copy of the interpreter. Of course, PHP scripts can manipulate files and databases to effect state changes but, from a programming perspective, the objects created by one request are destroyed when that request completes and therefore are not available to any subsequent request. These semantics have implications for the Ice extension as well; for example, it means that the run time creates a new communicator for each PHP request. (Ice 3.4 introduced enhancements in the Ice extension that allow a communicator to be used in multiple PHP requests, but we are not using this feature in the chat demo. See the [Ice manual](#) for more information.)

For applications that don't use a database (or don't want to incur the overhead of database operations), PHP supports a session facility that enables scripts to preserve state between requests. The special variable `$_SESSION` is a hash map in which a script can store values for retrieval in subsequent invocations.

Don't confuse a PHP session with the chat application's notion of a session, where a client using the pull model must log the user into the chat server via a session factory object and obtain a proxy for a new chat session. In order to use the designated chat session proxy in multiple PHP requests, our middle tier saves the proxy using a PHP session. More precisely, the scripts store a stringified proxy in the PHP session because an Ice proxy object cannot be stored directly. Consequently, each PHP request involves the extraction of the stringified proxy from the PHP session and conversion back into a proxy object.

Our middle tier consists of two files:

- `ChatI.php` is the entry point for requests from the browser. Its responsibilities include reading parameters, managing the HTTP session, and encoding the responses.
- `SessionI.php` implements the application logic; this is where all of the invocations on the back-end chat server are performed.

We'll review notable selections of code from both files, beginning with `ChatI.php`. If you'd like to see the complete source code, you'll find these files in the `client/php` subdirectory of the source distribution.

The Entry Point

Since the JavaScript code in the browser expects all responses to be encoded using JSON, the script defines a simple helper function to encode the data:

```
// PHP
function printJson($data)
{
    $json = new Services_JSON();
    print($json->encode($data));
}
```

The `ChatI.php` script starts by verifying that the Ice extension for PHP is loaded. If the extension is not found, the script prints an error message that will be shown in the browser and also records the same message in the web server's error log:

```
// PHP
if(!extension_loaded("ice")) {
```

```

    printJson("IcePHP extension is not loaded. " .
              "Revise your IcePHP installation.");
    error_log("IcePHP extension is not loaded. " .
              "Revise your IcePHP installation.");
    exit(1);
}

```

Next the script attempts to locate the Ice installation directory and adds the appropriate directory to PHP's include path. If the search fails, the script logs an error and exits:

```

// PHP
$IceIncludePath = "";

// ...

if($IceIncludePath == "")
{
    printJson("IcePHP includes not found. " .
              "Revise your IcePHP installation.");
    error_log("IcePHP includes not found. " .
              "Revise your IcePHP installation.");
    exit(1);
}

$IincludePath = get_include_path();
if(!in_array($IceIncludePath, explode(PATH_SEPARATOR, get_include_path())))
{
    if($IincludePath != '')
    {
        $IincludePath .= PATH_SEPARATOR;
    }
    $IincludePath .= $IceIncludePath;
    set_include_path($IincludePath);
}

```

The chat demo code tries to avoid the need for users to manually configure PHP's include path by accommodating a variety of installation scenarios, including support for multiple platforms. You can likely skip this step if the requirements for your own applications are simpler.

The next step is to load the PHP files required by our chat client:

- Ice.php contains the definitions needed by the Ice run time
- Chat.php and ChatPolling.php are generated from our Slice definitions
- SessionI.php handles communications with the chat server

Note that we don't need to explicitly include Chat.php because it is included by PollingChat.php:

```

// PHP
require_once 'Ice.php';
require_once dirname(__FILE__) . '/SessionI.php';
require_once dirname(__FILE__) . '/PollingChat.php';

```

Now we can initialize the Ice run time:

```

$Iice = Ice_initialize();

```

We need a unique identifier in order to configure a PHP session. The JavaScript code must define the `id` parameter if it has already established a session; our script checks for the presence of this parameter and passes it to PHP's session facility if one is found:

```
if(isset($_POST['id'])) {
    session_id($_POST['id']);
}
```

We are sending the identifier as a `POST` parameter so that we can support multiple chat sessions in the same browser. This works well for testing purposes because it allows a user to easily open several chat sessions simultaneously. For applications without this requirement, using a browser cookie may be a better option so that each web client is limited to just one session.

Calling `session_start` restores the previous state of the PHP session if an identifier has been specified, otherwise it creates a new session. For security reasons, we prefix the function call with `@` to prevent PHP from reporting any errors to the browser in case of a failure, and we explicitly check whether the session started successfully by testing for the presence of the `$_SESSION` variable:

```
// PHP
@session_start();
if(isset($_SESSION)) {
    $_action = isset($_POST['action']) ? $_POST['action'] : 'none';
    $session = new Session($ice);
    switch($_action) {
        case 'login': ...
        case 'logout': ...
        case 'send': ...
        case 'getUpdates': ...
        case 'getInitialUsers': ...
        default: {
            printJson("InvalidActionException");
            break;
        }
    }
}
```

The `action` parameter specifies the command that the JavaScript code wants to execute and is equivalent to an operation name in an Ice invocation. After retrieving the action, the script instantiates a `Session` object that encapsulates the application logic and then translates the action into a method call on `Session`:

```
// PHP
case 'login': {
    $userName = stripslashes(isset($_POST['username']) ?
        $_POST['username'] : 'nobody');
    $userPassword = stripslashes(isset($_POST['password']) ?
        $_POST['password'] : 'nobody');
    printJson($session->login($userName, $userPassword));
    break;
}
case 'logout': {
    printJson($session->logout());
    break;
}
case 'send': {
```

```

        $message = stripslashes(isset($_POST['message']) ?
            $_POST['message'] : '');
        printJson($session->send($message));
        break;
    }
    case 'getUpdates': {
        printJson($session->getUpdates());
        break;
    }
    case 'getInitialUsers': {
        printJson($session->getInitialUsers());
        break;
    }
}

```

Some actions have parameters that the script extracts from the `POST` data prior to invoking the method. The return value of each `Session` method is transformed into the output of the script, encoded in JSON as usual.

Managing Sessions

The definition of the `Session` class in `SessionI.php` supplies methods for each of the `POST` actions described above and additional methods for tracking the state of the session:

```

// PHP
class Session
{
    function __construct($ice, $properties) { ... }
    function login($username, $password) { ... }
    function logout() { ... }
    function send($message) { ... }
    function getInitialUsers() { ... }
    function getUpdates() { ... }

    function setConnected($connected) { ... }
    function isConnected() { ... }
    function checkIsConnected() { ... }
}

```

The constructor checks whether the user has already established a session (i.e., whether the user is connected); if so, the constructor retrieves a stringified proxy for a `PollingChatSession` object from the PHP session's state and converts that into a proxy:

```

// PHP
function __construct($ice, $properties)
{
    $this->_ice = $ice;
    $this->_properties = $properties;
    if($this->isConnected()) {
        $this->_chatsession =
            $this->_ice->stringToProxy($_SESSION["chatsession"]);
        $this->_chatsession = $this->_chatsession->ice_uncheckedCast(
            "::PollingChat::PollingChatSession");
    }
}

```

The `login` method shows how our middle tier establishes a chat session and records state in the PHP session:

```
// PHP
function login($username, $password)
{
    $chatsessionfactory = $this->_ice->stringToProxy(
        $this->_properties->getProperty(
            "PollingChatSessionFactory.Proxy"));
    $chatsessionfactory =
        $chatsessionfactory->ice_uncheckedCast(
            "::PollingChat::PollingChatSessionFactory");
    $this->_chatsession =
        $chatsessionfactory->create($username, $password);
    $this->setConnected(true);
    return new JavaScriptSession(session_id());
}
```

The first step is to obtain a proxy for the session factory, which the script does by retrieving the property containing the session factory proxy, converting that string into a proxy by calling `stringToProxy`, and finally narrowing the resulting proxy to the appropriate interface type. (You can find the definition of the property `PollingChatSessionFactory.Proxy` in the file `config/config.phpclient`.) The script invokes `create` on the factory to establish the session, passing the user name and password arguments that the user previously entered into a browser form. A call to the `setConnected` helper function records our state in the PHP session, and the method finishes by returning an instance of a `JavaScriptSession` object that transfers the identifier of the new PHP session to the JavaScript code in the browser. As we discussed earlier, the JavaScript code must supply this identifier in the `id` parameter of each subsequent `POST` request.

The `setConnected` helper function is instructive because it demonstrates how our scripts manage their state in a PHP session. The function expects an argument of `true` when we establish a new chat session, and `false` when we terminate the session:

```
// PHP
function setConnected($connected)
{
    if($connected) {
        $_SESSION["chatsession"] =
            $this->_ice->proxyToString($this->_chatsession);
        $_SESSION["connected"] = "true";
    } else {
        $_SESSION["connected"] = "false";
        $_SESSION["chatsession"] = "";
        $this->_chatsession = 0;
    }
}
```

Upon establishing a new chat session, the function records two entries in the PHP session: `chatsession` holds the stringified proxy, and `connected` indicates whether the user is currently in a chat session. These entries are reset when `setConnected` is called with an argument of `false`.

The last `Session` method we'll examine here is `getUpdates`, which illustrates the "protocol" that the JavaScript code uses to interpret the results of our PHP scripts:

```
// PHP
function getUpdates()
{
    $this->checkIsConnected();
}
```

```

$update = $this->_chatsession->getUpdates();
$data = array();
$length = count($updates);
for($i = 0; $i < $length; $i++)
{
    $update = $updates[$i];
    $update->json_type = get_class($update);
    $data[] = $update;
}
return $data;
}

```

The method begins by verifying that the user is still connected: the `checkIsConnected` helper function examines the contents of the PHP session and raises an exception if a chat session is not currently established. Next, the method invokes `getUpdates` on the chat session proxy and then iterates over the results to add a field named `json_type` to each element. The JavaScript code requires this field, which defines the PHP class name for the element's type, in order to reconstruct the polymorphic array of chat updates. Note that the code re-indexes the array to ensure that JSON maps the array to a JSON array and not to a dictionary.

Browser Scripting

The JavaScript code in the browser uses the Prototype library to handle certain tasks. For example, after the user has successfully logged in, the code creates a `PeriodicalExecuter` object that invokes a callback at the specified frequency:

```

// JavaScript
updater = new PeriodicalExecuter(this.getUpdates, 3);

```

In this case, the `getUpdates` function will be invoked every three seconds. We have already seen the server-side implementation of `getUpdates`, now let's examine its JavaScript counterpart:

```

// JavaScript
getUpdates:function()
{
    var params = new Hash();
    params.set('id', sessionId);
    params.set('action', 'getUpdates');
    var opts =
    {
        contentType:'application/x-www-form-urlencoded',
        method:'post',
        encoding:'UTF-8',
        parameters:params,
        onComplete:function(transport) {
            var response = transport.responseText.evalJSON(true);
            for(var i = 0; i < response.length; i++) {
                switch(response[i].json_type) {
                    case "PollingChat_MessageEvent":
                        if(response[i].name != _username) {
                            var message = new Element('div').update(
                                "<div>" + formatDate(response[i].timestamp)
                                + " - <" + response[i].name + "> " +
                                response[i].message + "</div>");
                            addMessage(message);
                        }
                        break;
                }
            }
        }
    }
}

```

```

        ...
    }
}
}
new Ajax.Request('Chat.php', opts);
}

```

When invoked by the `PeriodicalExecuter`, this function issues an asynchronous Ajax request that is executed by our PHP handler. The parameters of the `POST` request include the PHP session identifier and the action to be executed. The browser invokes `onComplete` when the request completes; this function decodes the JSON-encoded response into an array of chat updates and iterates over them to apply each update to the user interface. As discussed in the previous section, each array element has a `jsonType` field that indicates its type and determines how the update is processed.

If you are interested in seeing the complete JavaScript source code for the chat client, you can find it in the file `client/php/js/chat.js` from the source distribution.

Java Applet Client

An applet is a Java application that runs inside a browser. Since applets can be downloaded from untrusted web sites, they normally run inside a sandbox that places many restrictions on their access to local resources. For example, by default an applet cannot manipulate the local file system, nor can it open a socket to any host other than the one from which it was downloaded. However, applets can still make use of the full range of features provided by Ice for Java, including IceSSL, AMI, and callbacks, within the limitations imposed by the sandbox.

Our applet chat client uses the push model and closely resembles the other push clients in many ways. This section discusses aspects of the code that differ from the standalone Java and .NET graphical clients, the most significant of which is the implementation of the user interface. The standalone Java program uses the Swing tool kit to create its user interface, whereas the applet behaves more like the other web clients and uses a combination of HTML and JavaScript in its interface.

JavaScript Integration

Using HTML and JavaScript to define the applet's user interface provides a better end-user experience and offers better integration with the browser. To accomplish this, we need a way to call applet code from JavaScript, and a way to call JavaScript code from the applet.

The browser makes all of an applet's public methods accessible to JavaScript code running in the web page that loaded the applet, so our chat applet must define as public all methods that need to be callable from JavaScript. For example, the JavaScript code shown below invokes the `setDefaultRouter` and `login` methods on the applet:

```

// JavaScript
// Call the login operation on the applet.
function login()
{
    document.ChatDemoApplet.setDefaultRouter(DefaultRouter);
    document.ChatDemoApplet.login($('txtUserName').value,

```

```

        $('txtPassword').value);
    }

```

The document variable is a global reference to the DOM object that the browser defines for use in JavaScript. The symbol ChatDemoApplet corresponds to the name attribute of the applet tag in HTML:

```

<APPLET id="applet" CODE="ChatDemoGUI.ChatDemoApplet.class"
  MAYSCRIPT
  NAME="ChatDemoApplet"
  HEIGHT=120
  WIDTH=240>
  <PARAM NAME="archive" VALUE="ChatDemoAppletGUI.jar">
  <p id="alttext">
    Your browser understands the &lt;APPLET&gt; tag but
    isn't running the applet for some reason.

    Your browser is completely ignoring the &lt;APPLET&gt; tag!
  </p>
</APPLET>

```

For the applet to be able to invoke JavaScript code, the applet tag must include the MAYSCRIPT attribute as shown above.

During our development of the applet client, we noticed some differences in behavior among various browsers with respect to calling JavaScript code from an applet. Specifically, with some browsers, calling JavaScript code from the AWT event loop thread causes a deadlock. As a work around, the applet uses a work queue with a dedicated thread for dispatching calls to JavaScript:

```

// Java
class JSWorkQueue extends Thread
{
    public void run()
    {
        while(true)
        {
            Runnable r = null;
            synchronized(this)
            {
                while(!_queue.isEmpty())
                {
                    try
                    {
                        wait();
                    }
                    catch(java.lang.InterruptedException ex)
                    {
                    }
                }
                assert !_queue.isEmpty();
                r = _queue.remove(0);
            }
            r.run();
        }
    }

    synchronized public void enqueue(Runnable r)
    {

```

```

        if(_queue.isEmpty())
        {
            notify();
        }
        _queue.add(r);
    }

    private java.util.List<Runnable> _queue =
        new java.util.LinkedList<Runnable>();
}

```

When the applet needs to call into JavaScript, it encapsulates the call in a `Runnable` object and invokes `enqueue` on the work queue; the `Runnable` is eventually executed by the queue's thread.

Now let's explore how we can invoke JavaScript code from an applet. The `JSObject` facility is our preferred mechanism, and in browsers that don't support `JSObject` we fall back to using the `AppletContext.showDocument` method. For example, the code below shows how the applet calls into JavaScript to announce that a new chat message has been received:

```

// Java
public void userSay(final String time, final String user,
                   final String message)
{
    _queue.enqueue(new Runnable()
    {
        public void run()
        {
            if(_jsObject != null)
            {
                try
                {
                    _jsObject.call("userSay",
                                   new String[] {time, user, message});
                }
                catch(Exception ex)
                {
                    printJavaScriptException(ex);
                }
            }
            else
            {
                try
                {
                    getAppletContext().showDocument(
                        new URL("javascript:userSay('" + time + "', '" +
                                user + "', '" + message + "')"));
                }
                catch(java.net.MalformedURLException ex)
                {
                    printJavaScriptException(ex);
                }
            }
        }
    });
}

```

And here is the corresponding JavaScript code being invoked by the applet:

```
// JavaScript
function userSay(timestamp, name, message)
{
    chatView.userSay(timestamp, name, message);
}
```

The JavaScript function `userSay` invokes a method of the same name on the `ChatView` JavaScript object, which updates the HTML-based user interface to include the new chat message. We would have preferred for the applet to invoke the `ChatView` object's method directly, but again we found this wasn't portable across all target browsers so we added this helper function to work around the problem.

Application Logic

As in the other push clients, our application logic is encapsulated in the `Coordinator` class. We won't review the class here because its implementation is quite similar to that of the standalone Java client, including its use of AMI and the `Glacier2` helper classes. You can find the source code for the applet in the `client/applet` subdirectory of the source distribution.

Silverlight Clients

Silverlight is Microsoft's solution for creating rich web applications using .NET technology. A Silverlight program consists of .NET code compiled into a DLL assembly and Extended Application Markup Language (XAML) that describes the user interface, packaged together into an archive with a .XAP extension. When a web server is configured with the proper MIME types, a user can open a .XAP file in a browser and the Silverlight plug-in executes it automatically.

Ice for Silverlight is a lightweight implementation of the Ice run time that supports synchronous and asynchronous invocations. Ice requests issued by a Silverlight program can use one of two transports: a regular socket connection to a `Glacier2` router, or an HTTP or HTTPS connection to a generic ASP.NET bridge that runs as a web server module.

The Silverlight chat clients are similar to their PHP counterpart in that they all use HTML, CSS, and JavaScript in their user interfaces. Another similarity with PHP is the reliance on a web server module, which is also required by the Silverlight pull client. The main difference between the Silverlight and PHP clients is the location of the application logic: the Silverlight clients concentrate their behavior in the browser, whereas in PHP the behavior is split between the JavaScript code in the browser and the PHP scripts in the web server.

Creating a Silverlight Application Object

An application object is the entry point for every Silverlight program. Its class must inherit from `System.Windows.Application` and perform whatever steps are necessary to initialize the application. The implementation of the application object is identical in both of our chat clients, so we are presenting it here before we discuss the details of each client in separate sections below.

The application object is defined in `App.xaml` and `App.xaml.cs`; its definition is split into two files because it is a .NET partial class, which is a class that is partially defined in code and partially in an XAML markup file. The constructor of our application object is the first method that the Silverlight

plug-in calls after loading the application. The method configures event handlers for starting and terminating the application:

```
// C#
public App()
{
    this.Startup += this.Application_Startup;
    this.Exit += this.Application_Exit;
    this.UnhandledException += this.Application_UnhandledException;
    InitializeComponent();
}
```

The `Application_Startup` handler is invoked once the Silverlight components are properly initialized. The handler's implementation simply instantiates the `Coordinator` object, which is the root component of the application and must be assigned to the `RootVisual` property of the application object:

```
// C#
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new Coordinator();
}
```

The implementations of `Application_Exit` and `Application_UnhandledException` are currently empty.

The Silverlight Pull Client

An Ice for Silverlight client is unable to receive callbacks when it uses the ASP.NET bridge to forward requests, so our pull client polls periodically to receive new events. This polling needs to be done carefully, however, because the Silverlight client must strive to maintain a responsive user interface at all times. To avoid any risk of blocking the UI event loop, the client polls using asynchronous invocations.

Source Files

The source code for the Silverlight pull client is located in the `client/sl/bridge/ChatDemo` subdirectory of the source distribution. The following files comprise the application code:

- `App.xaml` – XAML definition of the Silverlight application object.
- `App.xaml.cs` – C# implementation of the Silverlight application object.
- `Coordinator.xaml` – XAML definition for the coordinator.
- `Coordinator.xaml.cs` – C# implementation of the coordinator, which contains most of the application logic.
- `ChatUtils.cs` – Utility methods for validation and formatting.

The subdirectory `client/sl/bridge/ChatDemoWeb` contains supporting files:

- `index.html` – Home page for the Silverlight push client.
- `index.html.js` – JavaScript code used to initialize the application and manage the HTML layout.
- `chatview.html` – HTML page that contains the client's user interface.

- `resources/style.css` – Cascading style sheet that defines the look and feel of the user interface.

Now that we have outlined the general structure of the client, we can proceed to examine it in more detail.

The Coordinator

The coordinator is a Silverlight component that implements the application logic of our chat client. It is also a scriptable object, meaning it offers an API that can be called directly from JavaScript code. Like the application object, the coordinator is a .NET partial class whose XAML definition is shown below:

```
<!-- XAML -->
<UserControl x:Name="parentCanvas"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="Page_Loaded"
    x:Class="ChatDemo.Coordinator"
    Width="0"
    Height="0"
    Background="White">
    <UserControl.Resources>
        <Storyboard x:Name="Timer"/>
    </UserControl.Resources>
</UserControl>
```

As you can see, XAML is based on XML; the root of our definition is a `UserControl` element that introduces a new Silverlight component. The XAML definition is associated with its .NET partial class via the `x:Class` attribute, and the nested `Storyboard` resource is used as a timer that can directly invoke methods from the event loop thread.

Now let's continue with the application logic implemented in `Coordinator.xaml.cs`. The class definition shows how to declare a scriptable partial class:

```
// C#
[ScriptableType]
public partial class Coordinator : UserControl
```

The `ScriptableType` attribute makes it possible for JavaScript code to use all public properties, methods, and events of this class.

The constructor invokes `InitializeComponent` (which is required for all Silverlight components) and then registers itself as a scriptable object with the name `ChatCoordinator`. By calling `RegisterScriptableObject`, we are making the coordinator instance available to JavaScript code:

```
// C#
public Coordinator()
{
    InitializeComponent();
    HtmlPage.RegisterScriptableObject("ChatCoordinator", this);
}
```

Silverlight also makes it possible for managed code to invoke JavaScript methods, and we lay the groundwork for using this feature in the `Page_Loaded` event handler:

```
// C#
public void Page_Loaded(object o, EventArgs e)
{
    _chatView = HtmlPage.Window.CreateInstance("ChatViewProxy");
}
```

This method is invoked when the enclosing HTML page is loaded. Its purpose is to create an instance of the JavaScript type `ChatViewProxy`, which is responsible for managing the client's HTML-based user interface. The return value of `CreateInstance` is a `ScriptObject` (a Silverlight wrapper around the JavaScript object) that we will use to update the user interface after receiving new chat events.

Logging In

As an example of how to integrate JavaScript and Silverlight, let's consider the process of logging the user into the chat server. The user starts by entering a user name and password and then clicking the `Login` button created by the following HTML code:

```
<!-- HTML -->
<div id="loginForm">
  <div class="line">
    <label for="txtUserName">Username</label>
    <input class="lineInput" type="text" id="txtUserName" tabindex="1"
      onKeyDown="passwordFocus(event);"/>
  </div>
  <div class="line">
    <label for="txtPassword">Password</label>
    <input class="lineInput" type="password" id="txtPassword" value=""
      tabindex="2"
      onKeyDown="loginOnEnterPressed(event);"
      onBlur="focusId('txtUserName');"/>
  </div>
  <div class="line">
    <div class="loginButtonWrapper">
      <a class="button" href="#" onclick="login();">
        <span>Login</span>
      </a>
    </div>
  </div>
</div>
```

We are not using a traditional HTML form here because some browsers automatically submit the form when the user hits the Enter key. To provide a better user experience, the code shown above uses JavaScript events: hitting Enter in the `txtUserName` field changes the focus to the `txtPassword` field, and hitting Enter in `txtPassword` (or clicking the `Login` button) executes the `login` function.

The `login` function composes the URL of the ASP.NET bridge that runs in the web server and passes it to the coordinator object. It also supplies the coordinator with a proxy for the session factory object (relative to the bridge), and then instructs the coordinator to establish a session using the user name and password from the HTML text fields:

```

// JavaScript
var PollingChatSessionFactory =
    'PollingChatSessionFactory:tcp -h 127.0.0.1 -p 10001';

function login()
{
    var sl = getSlControl();
    var pos = location.href.lastIndexOf("/");
    var loc = location.href;
    if(pos > 0) {
        loc = location.href.substr(0, pos)
    }
    sl.Content.ChatCoordinator.setBridgeUri(loc + "/IceBridge.ashx");
    sl.Content.ChatCoordinator.setSessionFactoryEndpoints(
        PollingChatSessionFactory);
    sl.Content.ChatCoordinator.login($('txtUserName').value,
        $('txtPassword').value);
}

```

The `getSlControl` function simply wraps a call to `document.getElementById("SilverlightControl");` the return value allows `login` to navigate to the coordinator object that was previously registered as the symbol `ChatCoordinator` and invoke methods on it. For example, consider the implementation of the `Coordinator.login` method:

```

// C#
public void login(string name, string password)
{
    _chatView.Invoke("setState", "Connecting");

    try {
        _communicator = Ice.Util.initialize();

        PollingChat.PollingChatSessionFactoryPrx sessionFactory =
            PollingChat.PollingChatSessionFactoryPrxHelper.uncheckedCast(
                _communicator.stringToProxy(_sessionFactoryEndpoints));

        AMI_ChatSessionFactory_createI callback =
            new AMI_ChatSessionFactory_createI(this, name);
        sessionFactory.create_async(callback.createSessionResponse,
            createSessionException, name,
            password);
    }
    catch(Ice.LocalException ex) {
        _chatView.Invoke("setError", "<div>" + ex.ToString() + "</div>");
    }
}

```

Recall that `_chatView` is a `ScriptObject` that refers to the JavaScript object that manages the HTML user interface. The `login` method begins by invoking the JavaScript function `setState` to notify the user that the login process has begun. The method then initializes a communicator, obtains a proxy for the session factory using the stringified proxy that the JavaScript code supplied, and asynchronously invokes an Ice operation to create a new session. It is very important that we use an asynchronous invocation here because `login` is called from Silverlight's event loop thread and we cannot risk blocking.

If you're familiar with Ice for .NET's old mapping for asynchronous operations, you would have expected the code to pass the callback object as the first argument to `create_async`, followed by the user name and password. The Silverlight mapping differs from the C# mapping in this respect, in that the Silverlight mapping expects the caller to supply two delegates instead of a single callback object. Ice invokes the first delegate upon success and the second delegate if an exception occurs. (The new AMI mapping introduced in Ice 3.4 incorporated this idea and added some additional improvements.)

Assuming the invocation of `create_async` succeeds, we eventually arrive in the `Coordinator.createSessionResponse` method with a proxy for the newly-created session:

```
// C#
public void createSessionResponse(
    string username,
    PollingChat.PollingChatSessionPrx session)
{
    Dispatcher.BeginInvoke(delegate() {
        _connected = true;
        _username = ChatUtils.formatUsername(username);
        _session = session;
        _session.getInitialUsers_async(getInitialUsersResponse,
                                       getInitialUsersException);
        _chatView.Invoke("setState", "Connected");
    });
}
```

As we saw earlier in the C# implementation, the Silverlight client also uses `Dispatcher.BeginInvoke` to schedule a delegate for execution in the event loop thread. The delegate begins another asynchronous invocation, this time to obtain the current users in the chat room, and updates the user interface to indicate that the user has logged in successfully.

In effect, the client has started a "chain reaction" where the response delegate for an asynchronous request issues another asynchronous request. This process started in the `login` method and continued in `createSessionResponse`. Now, the delegate for `getInitialUsers` prepares the client's central update loop:

```
// C#
private void getInitialUsersResponse(String[] users)
{
    Dispatcher.BeginInvoke(delegate() {
        for (int i = 0; i < users.Length; ++i) {
            _chatView.Invoke("addUser", users[i]);
        }
        getUpdates(null, new EventArgs());

        Timer.Duration = TimeSpan.FromSeconds(3);
        Timer.Completed += new EventHandler(this.getUpdates);
        Timer.Begin();
    });
}
```

After invoking the JavaScript function `addUser` once for each user in the list, the delegate schedules a timer to invoke the `getUpdates` event handler after three seconds. This handler calls the Ice operation of the same name (asynchronously, of course) in order to retrieve any queued chat events

and update the user interface. However, rather than waiting for the timer interval to elapse before polling for chat events, the delegate calls the `getUpdates` handler directly.

Bearing in mind that `getUpdates` is called from Silverlight's event loop thread, its only responsibility is starting the asynchronous invocation:

```
// C#
private void getUpdates(object sender, EventArgs e)
{
    _session.getUpdates_async(getUpdatesResponse, getUpdatesException);
}
```

This method has the signature expected of an event handler, but has no use for the arguments. Finally, the `getUpdatesResponse` delegate processes new chat events:

```
// C#
private void getUpdatesResponse(PollingChat.ChatRoomEvent[] events)
{
    Dispatcher.BeginInvoke(delegate() {
        for(int i = 0; i < events.Length; ++i) {
            if(events[i] is PollingChat.MessageEvent) {
                PollingChat.MessageEvent e =
                    (PollingChat.MessageEvent)events[i];
                if(!_username.Equals(e.name,
                    StringComparison.CurrentCultureIgnoreCase)) {
                    _chatView.Invoke(
                        "userSay",
                        ChatUtils.formatTimestamp(e.timestamp), e.name,
                        e.message);
                }
            } else if(events[i] is PollingChat.UserJoinedEvent) {
                ...
            } else if(events[i] is PollingChat.UserLeftEvent) {
                ...
            }
        }
        Timer.Begin();
    });
}
```

This delegate schedules a nested delegate for execution by the event loop thread. The code iterates over the events and processes each one according to its type. For the sake of brevity we have only shown the code for `MessageEvent`; the other event types are handled similarly and call back into JavaScript code to update the user interface. In the case of `MessageEvent`, the code explicitly ignores any message that originated from the current user, since those messages have already been posted to the user interface. The last step is to restart the timer so that `getUpdates` is called again.

The Silverlight Push Client

This Silverlight push and pull clients are similar in many ways. Rather than repeat much of the previous discussion, we will concentrate in this section on the unique aspects of the push client. You can find the source code for the push client in the `client/sl/callback/ChatDemo` and `client/sl/callback/ChatDemoWeb` subdirectories of the source distribution.

Logging In

As in the pull client, the login process begins in an HTML form that executes some JavaScript code. This code differs somewhat from the pull client, however, in that now we must configure the coordinator object with a proxy for the Glacier2 router:

```
// JavaScript
var DefaultRouter = "Glacier2/router:tcp -p 4502 -h 127.0.0.1";
// Call the login operation using a scriptable Cs method.
function login()
{
    var sl = getSlControl();
    sl.Content.ChatCoordinator.setDefaultRouter(DefaultRouter);
    sl.Content.ChatCoordinator.login($('txtUserName').value,
        $('txtPassword').value);
}
```

Similarly, the coordinator's implementation of the `login` method differs from its pull client counterpart because of its need to establish a session with the router, but before we can do that we need to initialize a communicator with appropriate configuration properties. Since we are using the socket transport we must supply a value for `Ice.Default.Router`:

```
// C#
// Configure properties needed by Ice for Silverlight
Ice.InitializationData initData = new Ice.InitializationData();
initData.properties = Ice.Util.createProperties();
initData.properties.setProperty("Ice.Default.Router", _defaultRouter);
initData.properties.setProperty("Ice.FactoryAssemblies",
    "ChatDemo,version=1.1.0.0");
// Create the communicator.
_communicator = Ice.Util.initialize(initData);
```

The client's initialization process continues with the creation of a Glacier2 session, which we execute in a separate thread to avoid blocking the application's event loop:

```
// C#
System.Threading.Thread loginThread = new System.Threading.Thread(() => {
    try {
        Ice.RouterPrx defaultRouter = _communicator.getDefaultRouter();
        if(defaultRouter == null) {
            Dispatcher.BeginInvoke(delegate() {
                lock(this) {
                    _chatView.Invoke(
                        "setError",
                        "Login failed: Ice.Default.Router not set.");
                }
            });
            return;
        }
        router = Glacier2.RouterPrxHelper.checkedCast(defaultRouter);
        if(router == null) {
            Dispatcher.BeginInvoke(delegate() {
                lock(this) {
                    _chatView.Invoke(
                        "setError",
                        "Login failed: Ice.Default.Router is not a " +
                        "Glacier2 router.");
                }
            });
        }
    }
});
```

```

        }
    });
    return;
}

_session = Chat.ChatSessionPrxHelper.checkedCast(
    router.createSession(name, password));
lock(this) {
    _connected = true;
}
long sessionPingPeriod = router.getSessionTimeout() * 1000 / 2;
Dispatcher.BeginInvoke(delegate() {
    setConnected(name, sessionPingPeriod);
});
}
catch(Glacier2.CannotCreateSessionException ex)
{
    // ...
}
//...
});
loginThread.Start();

```

The remainder of the client's initialization process is performed by the `setConnected` and `join` methods, which update the user interface to reflect our connected status, create a timer that "pings the session" to keep it alive, activate an object adapter to receive chat updates, and register our callback object with the chat server. Again some of these steps are delegated to a separate thread:

```

// C#
private void setConnected(string name, long timeout)
{
    lock(this) {
        _connected = true;
        _username = ChatUtils.formatUsername(name);
        _chatView.Invoke("setState", "Connected");
        _timer.Interval = TimeSpan.FromMilliseconds(timeout);
        _timer.Tick += new EventHandler(pingSession);
        _timer.Start();
        join();
    }
}

private void join()
{
    System.Threading.Thread joinThread =
        new System.Threading.Thread(() => {
            try {
                Ice.ObjectAdapter adapter =
                    _communicator.createObjectAdapterWithRouter(
                        "ChatDemo.Client", router);
                Ice.Identity callbackId = new Ice.Identity();
                callbackId.name = "ChatDemoCallback";
                callbackId.category = router.getCategoryForClient();
                Chat.ChatRoomCallbackPrx callback =
                    Chat.ChatRoomCallbackPrxHelper.uncheckedCast(
                        adapter.add(new ChatRoomCallbackI(this),
                            callbackId));
                _session.setCallback(callback);
            }
            catch(Exception ex) {

```

```

        Dispatcher.BeginInvoke(delegate() {
            if(ex is Ice.UnknownLocalException) {
                Ice.UnknownLocalException unknowEx =
                    (Ice.UnknownLocalException)ex;
                connectionLost("...");
            }
            else {
                connectionLost("...");
            }
        });
    }
    });
    joinThread.Start();
}

```

Receiving Chat Updates

The callback implementation for the Silverlight push client is straightforward and nearly identical to the C# version. As you can see, the callback's methods simply delegate to the coordinator object:

```

// C#
public class ChatRoomCallbackI : Chat.ChatRoomCallbackDisp_
{
    public ChatRoomCallbackI(Coordinator coordinator)
    {
        _coordinator = coordinator;
    }

    public override void init(string[] users, Ice.Current current)
    {
        _coordinator.initEvent(users);
    }

    public override void send(long timestamp, string name, string message,
        Ice.Current current)
    {
        if (!name.Equals(_coordinator.getUsername(),
            StringComparison.CurrentCultureIgnoreCase))
        {
            _coordinator.userSayEvent(timestamp, name, message);
        }
    }

    public override void join(long timestamp, string name,
        Ice.Current current)
    {
        _coordinator.userJoinEvent(timestamp, name);
    }

    public override void leave(long timestamp, string name,
        Ice.Current current)
    {
        _coordinator.userLeaveEvent(timestamp, name);
    }

    private Coordinator _coordinator;
}

```

Summary

Developing a user interface for a networked application requires carefully balancing human factors, the requirements of the application, and the capabilities of your platforms and tools. In this article we presented six implementations of a client for our chat system: the Java and C# graphical programs are traditional desktop applications, while the PHP, the Java Applet and the two Silverlight clients use web-based interfaces. The primary design goal for all of these clients is a responsive user interface. Achieving that goal required us to use an asynchronous architecture that adds a bit more code and some extra complexity, but is an acceptable tradeoff.

Although we could not show every line of code for each of the chat clients here, we selected aspects of the implementation that will aid you in developing your own interactive Ice applications. We encourage you to download the source code for the chat system, and we look forward to discussing it with you on our forums.