



The Jabber message protocols

In this chapter

- The Jabber message protocol and how it works
- The basic design of a Java Jabber client
- Source code for implementing a Java Jabber client
- Creating clients for testing our Jabber server's message protocol support

Messaging is the heart and soul of every IM system. The Jabber <message> protocols provide a simple yet powerful framework for sending and receiving messages. In this chapter, we will discuss the Jabber message protocol and how it works. To demonstrate, we will create a basic Jabber client that can send and receive messages through the Jabber server we developed in the last chapter.

4.1 *Messaging is the heart of IM*

Sending messages is the primary responsibility of the Jabber system. Jabber supports six primary types of <message> packets: normal, chat, groupchat (conferences), headline, error, and out-of-band messages. Each uses a different model of communication and is best suited for different situations. The following table summarizes the various message options.

Table 4.1 Jabber message types and messaging model.

Message Style	Type	Model	Typical interface
Normal	normal	Email-like messages (default)	Message editor
Chat	chat	One-on-one online chat	Line by line chat
Groupchat	groupchat	Online chatroom	Line by line chat
Headline	headline	Scrolling marquee message	“Stock Ticker”
Error	error	Message error occurred	Alert dialog box
Out-of-Band	X Extension jabber:x:oob	Direct client to client file exchange. Defined in an X extension in <message> element.	Napster/FTP

The first five message types fall within the Jabber <message> protocol. Each sends its contents within the <message> element. These are the most common types of messages sent in Jabber systems. In this chapter, we’ll build a client that supports them.

Out-of-band messages provide a mechanism for clients to directly exchange data (typically files). The out-of-band protocol uses the Jabber server to exchange information about how the clients will talk to each other (usually by sending a web URL for downloading the file).

You can send out-of-band tags as either an X extension within a <message> element, or as an Info/Query packet¹. We’ll briefly cover the X extension version of the out-of-band message in this chapter. The IQ protocol is explained in detail in chapter 6, and the exact usage of the out-of-band IQ extension is detailed in appendix A.

¹ X Extensions are simple Jabber packets in an <x> element. They provide an extension mechanism for adding custom content to standard Jabber packets. I’ll discuss this in more detail later in the chapter.

4.2 The message protocol

The message protocol is extremely simple: message packets are sent from a sender to a recipient. By default, there is no acknowledgement when the recipient receives the message. If a message is sent and the recipient is not reachable, the server is obliged to store the message and deliver it when the recipient becomes available,² a process referred to in messaging systems as *store and forward*.

A basic message packet consists of a `<message>` element with the typical Jabber `from`, `to`, and `id` packet attributes. The message packet supports four standard subelements³ shown in table 4.2.

Table 4.2 The sub-packets allowed within a `<message>` packet.

Sub-Packet	Description
<code><subject></code>	Indicates the subject of the message similar to the subject field in an email message.
<code><thread></code>	A client generated identifier to help track messages belonging to a single “thread” of conversation.
<code><body></code>	The message body is enclosed in this element.
<code><error></code>	If an error occurred, the standard Jabber error packet is enclosed in the message.

Let’s take a look at the XML for a complete message packet. This message is being sent from “iain” to “smirk.” Most message packets do not contain an `id` attribute (it is optional).

Sample message packet

```
<message from='iain@shigeoka.com/work'
  to='smirk@jabber.org/home'
  id='messageid1'>
  <thread>threadid_01</thread>
  <subject>The message's subject</subject>
  <body>The text in the message body</body>
</message>
```

² In the Jabber server developed in the previous chapter, I cheated a little and we simply dropped messages addressed to someone that is offline. This flaw will be fixed in chapter 7 when we create user accounts on the server.

³ There are several X extensions that are also supported within a `<message>` packet. I’ll cover what an X extension is and show an example when I discuss the out-of-band X extension later in this chapter.

The `<thread>` packet is used to keep different threads of messages together.⁴ In this example, our thread ID is `threadid_01`. All messages sent between clients with the same thread ID will be displayed together⁵. In most graphical user interfaces (GUIs) this would be shown in a line-by-line chat interface. This allows you to chat with several people at once and keep each conversation separate.

When clients send messages to servers, the sender is implied to be the client's Jabber ID, and the recipient is assumed to be the server if no recipient is specified. Some Jabber servers may not allow you to send messages with a sender address that does not match the sender's session address. A perfectly valid (and typical) message sent to the server is:

```
<message to='smirk@jabber.org' >
  <body>howdy</body>
</message>
```

The server fills in implied fields for final delivery as shown in this example:

```
<message from='iain@shigeoka.com/work'
  to='smirk@jabber.org' >
  <body>Howdy</body>
</message>
```

4.2.1 Normal messages

The default message type is a *normal message*. These messages are typically created and displayed using interfaces similar to that used in email applications (figure 4.1). Like email, normal messages are sent to Jabber users who aren't necessarily online. These messages tend to be longer than other message types and resemble letters or memos.⁶

Message packets that do not contain a `type` attribute are considered normal messages (figure 4.1). In addition, you can explicitly indicate a message is a normal message by setting the `type` attribute to `normal` as shown in the following example.⁷

⁴ The `<thread>` packet is an aid to the Jabber client for correctly displaying related messages. It is not required, however, and clients should be able to display messages missing the `<thread>` packet.

⁵ The `id` attribute indicating the packet ID is also used to link related packets. However, the packet ID links request packets to response packets, or any packet and its associated error messages. The Info/Query protocols covered in chapter 6 rely heavily on the packet ID for its request-response process.

⁶ Typically, normal messages contain static content intended for offline delivery or more formal communications. Chat messages (covered in the next section) are intended for short messages where the user may or may not be online (all message types will be stored offline and delivered later). Think of normal messages as letters, while chat messages are Post-It notes.

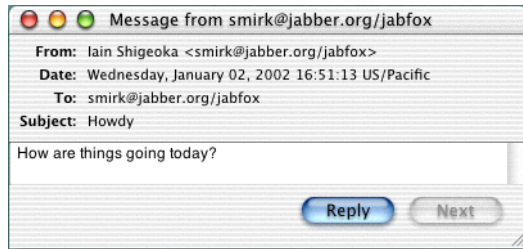


Figure 4.1
A typical normal message being displayed in the JabberFOX client (jabberfox.sourceforge.net).

Sample normal message packet

```
<message from='iain@shigeoka.com/work'
  to='smirk@jabber.org/home'
  id='messageid2'
  type='normal'>
  <thread>threadid_02</thread>
  <subject>The message's subject</subject>
  <body>The text in the message body</body>
</message>
```

It is typical for Jabber client applications to offer users the ability to start chatting with the sender of a normal message.

4.2.2 Chat messages

Jabber users send chat messages back and forth to other users who are online at the same time they are. These messages tend to be short and conversational, like the type of communication you do over a telephone. Chat messages are typically displayed in a line-by-line interface.⁸ When you write a chat line-by-line interface, you must place a copy of the messages you send into the chat window so the user can see both sides of the conversation.

Chat messages (figure 4.2) must have their `type` attribute set to `chat`. In addition, the message should contain a `<thread>` subelement. Jabber clients link messages into a threaded conversation using the `<thread>` ID. All chat messages that belong to a single conversation should use the same `<thread>` ID. It is common for the `<subject>` to be omitted in chat messages.

⁷ There are few good reasons to force the normal message type. If you're sending a normal message, leave the `type` attribute out. It's more efficient. The only reason I can see to force it is if your software is not flexible enough to send packets without a `type` attribute.

⁸ There are many innovative ways of displaying chat messages. Some clients display them as "thought bubbles" above animated cartoon characters, while others may use virtual reality or text-to-speech software to enhance the chat experience.

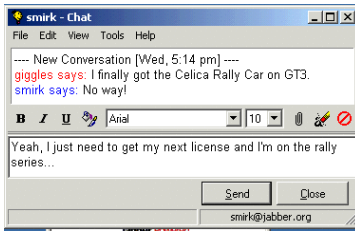


Figure 4.2
A typical chat message being displayed and edited in the Jabber Instant Messenger client (www.jabber.com).

Sample chat message packet

```
<message from='iain@shigeoka.com/work'
  to='smirk@jabber.org/home'
  id='messageid3'
  type='chat'>
  <thread>threadid_03</thread>
  <body>The text in the message body</body>
</message>
```

Chat is useful for conducting one-on-one online conversations. When you need to converse with many people at once, you need to use groupchat messages.

4.2.3 Groupchat messages

Groupchat messages are similar to chat messages but they are designed to support online conversations within groups. Instead of a one-on-one conversation like chat, groupchat allows many users to send and receive messages from an entire group of people.⁹ Everyone participating in the group, including the sender, receives a copy of the message.

When creating your groupchat user interface (figure 4.3), your interface should send groupchat messages to the group and update the groupchat interface with incoming messages. Groupchat servers automatically send groupchat messages to participants (including the sender). This feature relieves you from manually copying your outgoing messages to the groupchat window.¹⁰

A groupchat server manages groupchat conversations. In most Jabber servers, the groupchat server will be built into the Jabber server although it is possible to design a separate Jabber server that functions solely as a groupchat server. Groupchat conferences may be enhanced, supplemented, or replaced in the future

⁹ Jabber groupchats are often called conferences, chatrooms, or forums.

¹⁰ In chat message interfaces you must manually copy your outgoing messages into the chat window so the user sees both sides of the conversation. The server will not send you a copy of your own messages like it does in groupchat messaging.

by conferences using the `jabber:iq:conference` IQ extension protocol. I expect that the two will coexist in future Jabber servers. See the `jabber:iq:conference` IQ extension specification in this book located in appendix A.



Figure 4.3
A typical chat message being displayed and edited in the Jabber Instant Messenger client (www.jabber.com). Notice that the presence of participants is shown on the right.

Groupchat message packets are like chat packets except the type attribute is set to `groupchat`. Notice that you are sending messages to the groupchat server and not directly to another Jabber user. Groupchat groups use special Jabber IDs. The standard format for a groupchat group address is: `[group name]@[groupchat server]/[user nickname]`

Users can choose an arbitrary groupchat nickname for each group they join. It doesn't necessarily have anything to do with their regular Jabber user name. For example, we have a user name of "iain," with a Jabber ID of "iain@shigeoka.com/work." We want to send a message to the groupchat group named "java-users" on the groupchat server `conference.shigeoka.com`. We've already joined the group using the nickname "hacker." Our client sends a `<message>` packet to the group (`java-users@conferences.shigeoka.com`) that looks like the following:

Sample groupchat message outgoing packet

```
<message from='iain@shigeoka.com/work'
  to='java-users@conference.shigeoka.com'
  id='messageid4'
  type='groupchat'>
  <thread>threadid_04</thread>
  <body>The text in the message body</body>
</message>
```

The groupchat server receives the message and sends it to all members of the java-users group including our Jabber client.

Sample groupchat message incoming packet

```
<message from='java-users@conference.shigeoka.com/hacker'  
  to='iain@shigeoka.com/work'  
  id='messageid4'  
  type='groupchat'>  
  <thread>threadid_04</thread>  
  <body>The text in the message body</body>  
</message>
```

One of the great things about this design is that the other members of the group never see my real Jabber ID. The only information they know is that the messages are coming from `java-users@conference.shigeoka.com/hacker`. This prevents people from hanging out on the groupchat server and scraping Jabber IDs from the groups for spam lists or stalking users outside of the conference.

There is something missing from the groupchat message protocol, though. If you were paying close attention to the outgoing groupchat packet example, you'll notice that there is no information in it telling the server that I'm using the nickname "hacker." Nor can the conference server determine who is in the conference just from the message itself.

Mapping Jabber IDs to nicknames, managing conference membership, and other administrative issues concerning Jabber groupchat groups are handled using the Jabber presence protocols.¹¹ An advanced form of groupchat called conferencing is being proposed and uses the IQ protocols.¹²

4.2.4 *Headline messages*

Headline messages are Jabber messages designed for display in scrolling marquees, status bars, or other client interfaces designed for streaming information. It is common for automated chatbot services to generate headline messages concerning current events and news such as weather reports, severe weather alerts, and stock quotes.

Headline messages use a type attribute of `headline` and typically don't require a `<thread>` or `<subject>` subelement.¹³

¹¹ Presence protocols are discussed in chapter 5 where we'll implement support for basic groupchat.

¹² Info/Query protocols are discussed in chapter 6. The conferencing protocol is included in appendix A.

¹³ There is an exception to the lack of `<subject>` tags in headline messages. When creating streaming news services, it is common to use the `<subject>` for a news subject line, `<body>` for the news article itself, and the oob X extension to provide a URL for more information. This is similar to RDF Site Summary (RSS) functionality (purl.org/rss/1.0/spec).

Sample headline message packet

```
<message from='quote-bot@stockbroker.com'
  to='iain@shigeoka.com/work'
  id='messageid5'
  type='headline'>
  <body>SUNW 10</body>
</message>
```

The last message type that is supported by the `<message>` packet is the standard Jabber error message.

4.2.5 Error messages

When you send a message, there is always a chance that something will go wrong or the recipient will refuse the message. The error message type is used to notify the sender that the message they sent has encountered problems. The error packet shown in listing 4.1 is the standard Jabber error packet that we covered in the last chapter.

Listing 4.1 Sample error message packet

```
Send: <message from='iain@shigeoka.com/home'
Send:         to='hotbabe@shigeoka.com'
Send:         id='messageid6'
Send:         type='normal'>
Send:   <subject>Doing anything tonight?</subject>
Send:   <body>Hi, how about a date!</body>
Send: </message>
Recv: <message from='hotbabe@shigeoka.com/jacuzzi'
Recv:         to='iain@shigeoka.com/home'
Recv:         id='messageid6'
Recv:         type='error'>
Recv:   <error code='400'>
Recv:     Go away!
Recv:   </error>
Recv: </message>
```

Notice that the message id attribute (`messageid6` in this example) is preserved from the original message to the error message. This allows the client to match up the message it sent with the error message it received. Remember that messages are normally sent one-way so you may have sent many other messages before receiving an error message response.

NOTE Error messages don't necessarily refer to the last message you sent. You must match error messages to their cause by examining the packet ID.

4.2.2 Out-of-band messages

The last type of standard Jabber message, an out-of-band message, isn't really a Jabber message at all. Instead, it is a message X extension that is sent inside of a standard Jabber `<message>` packet (usually a message of type `normal`).

An out-of-band message contains information, typically a URL, that clients can use to conduct a direct client-to-client data transfer that bypasses the normal client-server-client Jabber message routing. Jabber clients will typically implement this by running a web server or FTP server either separately or as part of the Jabber client. The out-of-band message then tells the downloading client what URL to use to hit the web/FTP server and download the file.

Out-of-band messages are typically used to arrange sending large files that would cause severe server bandwidth shortages were routed through the server. For example, you may want to add music trading to your Jabber client application. The chat and song searching functions can occur over the Jabber server, but transferring multimegabyte MP3 files through the server would quickly bring your server to its knees. Ideally these high bandwidth transfers can be done directly between the clients resulting in:

- Reduced server load
- Possibly faster transfers
- Support for streaming network broadcasts

OUT-OF-BAND SECURITY RISKS

Note that the advantages of out-of-band messaging don't come without a cost. Since clients must directly communicate with each other, the client's security and privacy can be violated in ways that are impossible when all communication occurs through the Jabber server. In addition, network issues such as getting through firewalls and proxy servers are multiplied when clients must act as file servers.

These issues lie beyond the Jabber standards so Jabber client developers are often left on their own when trying to create robust and secure out-of-band systems. In addition, your solution may not work with other Jabber clients unless everyone agrees on how the out-of-band transfers will take place. Out-of-band messaging is an area of great interest and debate in the Jabber community. Hopefully in the future, the protocols for carrying out an out-of-band transfer will be standardized and added to the Jabber standards.

Two protocols are involved in carrying out an out-of-band transfer. The first uses the oob X extension for exchanging URLs used for the transfer, the second uses the oob IQ protocol for initiating the transfer.

Two protocols are involved in carrying out an out-of-band transfer. The first uses the oob X extension for exchanging URLs used for the transfer, the second uses the oob IQ protocol for initiating the transfer. We'll cover the Info/Query protocol in chapter 6. Let's take a look at the X extension technique here. In order to understand how the out-of-band X extension works, we first need to understand X extensions in general.

X extensions

The Jabber designers know that although the Jabber packets can handle the majority of IM tasks, there will always be additional features that people would like to support. To keep the protocol extension process under control, the Jabber core protocols support X extensions.

An X extension is simply an `<x>` packet within the core Jabber packet types: `<message>`, `<presence>`, and `<iq>`. By making `<x>` packets a valid subelement of the core packets, you can comply with the Jabber DTD and create a valid Jabber packet that contains this mysterious `<x>` packet.

The `<x>` packet has no default sub-elements.¹⁴ To create an X extension, you must define a new namespace within the `<x>` packet, and then insert any XML information you want in the packet. The XML namespace ensures that you won't violate the validity of the resulting XML fragment. Sometimes it is easier to show an example than to try to explain. So let's take a look at the out-of-band X extension.

The out-of-band X extension

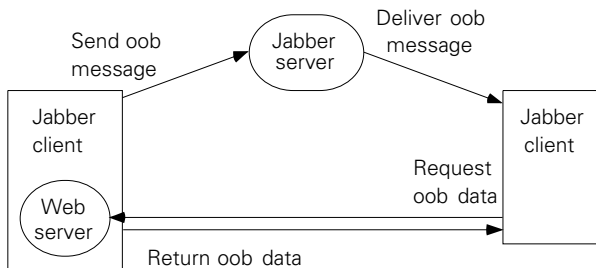
The out-of-band (oob) X extension is a standard Jabber X extension that allows you to specify an out-of-band transfer mechanism. Think of it as a URL passer.¹⁵ It resides in the `jabber:x:oob` namespace and contains two subelements:

- `<url>`—The URL describing the out-of-band transfer.
- `<desc>`—A text description of the data to be transferred.

¹⁴ For XML gurus, the `<x>` packet is defined in the parent packet DTDs so that it is a valid subpacket that conforms to the Jabber DTDs. However the `<x>` packet itself has no default subpackets allowing you to define its contents independently using an XML namespace.

¹⁵ The actual file exchange is initiated using the out-of-band IQ protocol `jabber:iq:oob`. You can learn more about the Info/Query protocol in chapter 6 and the specifics of the `jabber:iq:oob` protocol in appendix A.

The out-of-band X extension tells the receiving client where to get the file (figure 4.4). It does not actually contain the file (otherwise it would have passed through the server). In general, out-of-band transfers are an advanced feature typically found in only the more sophisticated Jabber clients

**Figure 4.4**

Out-of-band data is sent as an oob message through the Jabber server. The recipient must request the actual oob data directly from the sending client using some other protocol such as HTTP or FTP (here a web server and HTTP is used).

An out-of-band X extension in a message packet might be presented in a Jabber client with an email with attachments interface.

A message packet with out-of-band X extension

```

<message from='iain@shigeoka.com/work'
  to='iain@shigeoka.com/home'
  id='messageid8'
  type='normal'>
  <subject>Work files</subject>
  <body>Attached are some work files I may need</body>
  <x xmlns='jabber:x:oob'>
    <url>http://workserver/book.zip</url>
    <desc>Archive of my book</desc>
  </x>
</message>

```

In the interest of focusing on core Jabber protocols, we'll end our discussion of X extensions here. Although they are important parts of the Jabber protocols, X extensions are not essential for IM. By reading the reference section on the existing X extensions at the end of this book, it should be easy to determine what X extensions are important for your project.

The client software we develop in this book will not support out-of-band transfers.

4.2.3 Reality check: one message, many user interfaces

The six message types covered here represent the full range of messaging options available in a standard Jabber server. From this limited selection of message types, rich communication experiences have been created. In fact, from a

packet and protocol standpoint, the messages are all similar and it is perfectly valid to consider them all the same packet from a software standpoint.

The real difference in the Jabber message types lies not in the packet structure or protocols but in the interface that Jabber clients provide for the user to interact with the different message types. In fact, I believe that Jabber clients and the Jabber IM experience are driven by the user interface, not the quality of the technical implementation of the protocols.

The Jabber protocols are simple. The real challenge facing Jabber developers is to create friendly, enjoyable, exciting, and productive Jabber user experiences using them. To help you explore these issues, we'll create a bare-bones Jabber client that you can use for your own user interface experiments.

4.3 Java Jabber client

Jabber servers have a fairly well-defined role handling Jabber connections and properly responding to requests. Jabber clients on the other hand can appear in a bewildering variety of forms. The most common is a stand alone IM client that operates similarly to well-known IM clients such as AIM or ICQ.

Jabber clients don't have to be written that way, though. You can add Jabber client capabilities to your existing application offering your users built-in IM or simple chat features. Alternatively, many people are writing chatbots, Jabber client programs that act as servers in their own right, offering services on top of the Jabber system.

For example, you could write a chatbot that looks like another Jabber user on the Jabber server. If you send an IM to the chatbot, it might respond by telling you the local weather forecast or your bank account balance. The Jabber services provided by chatbots are similar to the heavily hyped web services programming model that uses XML over the Internet for similar purposes. Chatbots and other advanced uses for Jabber clients are covered in more depth in chapter 10.

Table 4.3 Typical types of Jabber client applications. Most users are familiar with the graphical user agent clients they use on their desktop. However, developers will probably find the most opportunities in developing chatbots and test clients.

Client type	Relationship with human "user"	Example applications
User agent	Tool for using Jabber	IM messaging application, chat feature in games
Chatbot	Provides "services" via Jabber IM	A weather service, a stock quote service
Test client	None	Standards compliance, stress testing, benchmarking

This section will show you the code for a basic test client. Throughout the rest of the book we'll use the client to test the Jabber server and to demonstrate client features of the Jabber protocols. In addition, you can use the Jabber client software developed here in your own Jabber client software projects.

4.3.1 Goals

As with the Jabber server software, we want the client to be simple, standards-compliant, easy to understand, and easy to modify. In addition to these goals, the client code is designed to be easily usable both in applications where it has a user interface, and those where it does not. Although most people's initial fascination with IM software lies in creating IM clients, I believe a few players will come to dominate this market just as they do for web and email clients. I anticipate that the real market for most developers of IM software is implementing Jabber functionality inside of other applications, in embedded systems, and offering Jabber chatbot services over the IM network.

The Jabber client developed in this book will not be particularly useful as a user-friendly IM client. The reason is practical. There simply is not enough room in this book to cover the source code needed to build a full-featured user interface with features such as extensive error checking, user customization, and help files that should be part of any user application.

User applications contain a bewildering amount of minor user interface details that are straightforward to implement, but are composed of a large quantity of mundane, tedious code. It would be a waste of paper to print the source code for all of that. In addition, it is exactly these details that will differentiate your Jabber client from another one. If you need a fully functional Jabber client to use with the Jabber server created in this book, you can either expand the Jabber client code to create a GUI, or simply download one of the many free Jabber clients that are available (see www.jabber.org to get started).

Although a full-blown user agent program is beyond the scope of this book, you should be able to create one from the source in this book. To keep things manageable, the client in this book will only support the Jabber features that I cover in detail in this book:

- *Messaging*—Sending and receiving Jabber IM.
- *Presence*—Sending and receiving presence information.
- *Info/Query*—Basic information exchange between Jabber entities. This is a broad area and I'll restrict myself to three protocols from the full set IQ protocols:

- *Roster Management*—Subscribing and maintaining your online presence status.
- *Registration*—Creating user accounts on open Jabber servers.
- *Authentication*—Logging in to a Jabber server.

4.3.2 The client design

Our Jabber client is an extremely simple piece of software. Right now, it needs to complete the following basic tasks:

- Connect to a Jabber server.
- Send an opening `<stream:stream>` tag.
- Send `<message>` packets.
- Receive and display `<message>` packets.
- Send a closing `</stream:stream>` tag.
- Exit.

To aid us in debugging and to let the user know what is happening, we also want the client software to indicate to the user what the status is of the Jabber session, and to provide a listing of all the raw data being sent between the client and server.

We will use the Model-View-Controller (MVC) design pattern (figure 4.5) to facilitate the use of the client software with graphical user interfaces (GUI) as well as in applications with a limited or absent user interface. The MVC design pattern is described by Buschmann et al¹⁶ as follows:

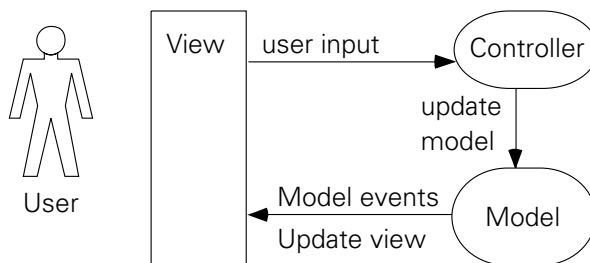


Figure 4.5 The Model-View-Controller design pattern separates the user's display (View), from interpretation of user inputs (controller), and the data and functionality of interest (Model).

¹⁶ Buschmann et al, *Pattern-oriented Software Architecture: a System of Patterns* (John Wiley & Sons, 1996), p. 125.

The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

In this book, we will focus on discussing the client model classes from the MVC pattern. In this chapter, the client model must manage the Jabber connection, keep a status model, and log the Jabber XML stream. The client model will reuse most of the server code covered in the previous chapter so this is not as much work as it sounds.

We'll create a rudimentary test harness around the model to show how to use it, and to drive our client/server tests. I don't want to place too much emphasis on the user interface aspects of the client code so the client won't have one. User interfaces are something I leave to you to design for your own needs and tastes.

I hope that you will be able to use this book's source code to create your own user interfaces and attach them to our client model. You can therefore build and control the look and feel of your Jabber client, while using the book code to manipulate Jabber functionality and data. In addition, if you don't need a user interface, you can build an application that directly manipulates the client model we build here.

With that said, let's take a look at the client model source code.

4.3.3 The client model

The client model classes will handle all of the Jabber responsibilities of the client. Recall that for this chapter, we want to open a Jabber stream, send messages, receive messages, and close the stream. We have already developed software that does these things as part of the server in chapter 3. We can simply reuse that code here to create a client model, as shown in figure 4.6.

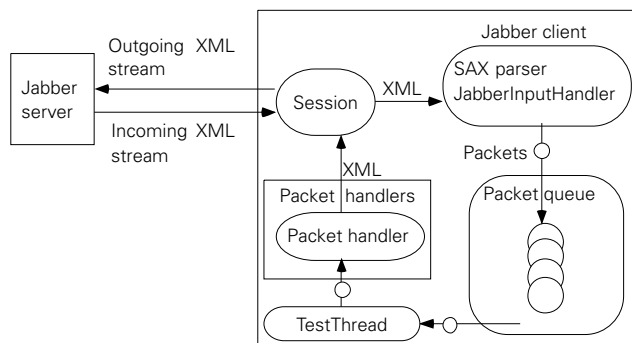


Figure 4.6 The client follows a similar architecture to the server, and reuses many of the server's classes.

We are not just reusing the server's design. Many of the main server classes, like the `JabberInputHandler` and `ProcessThread` are directly reused from the server. This makes the client-specific code compact enough that we can easily package it into a single model class, the `JabberModel`.

The `JabberModel`'s primary job is to make Jabber related tasks simple. Packet handling is carried out by a combination of `TestThread` actions and packet handling classes similar to that on the server. The `JabberModel`'s basic operations are outlined in the sequence diagram shown in figure 4.7.

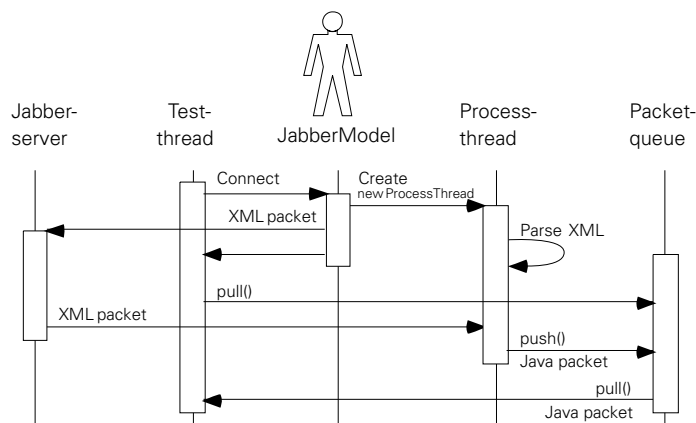


Figure 4.7 The `TestThread` (replacing the server's `QueueThread`) uses the `JabberModel` to create a `ProcessThread` and its associated packet handling classes. The `TestThread` operates by sending packets using the `JabberModel` and pulling responses from the `PacketQueue`. The `TestThread` also hands Packets to packet handling classes for special handling.

The `JabberModel` class constructor

```

public class JabberModel {

    JabberModel(TestThread qThread) {
        packetQueue = qThread.getQueue();
        qThread.addListener(new OpenStreamHandler(), "stream:stream");
        qThread.addListener(new CloseStreamHandler(), "/stream:stream");
        qThread.addListener(new MessageHandler(), "message");
    }

    Session session = new Session();
    PacketQueue packetQueue;
  
```

The `JabberModel` constructor should look familiar. We saw similar code in the `Server` class from chapter 3 to set up the `QueueThread` with `PacketListeners`. The `JabberModel` similarly configures the `TestThread` with `PacketListeners` to handle

incoming packets. The packet-handling interface to the `TestThread` is closely modeled on the `QueueThread`. The client uses different versions of the `PacketListener` classes than the server because we want it to exhibit different behavior when packets arrive. We'll cover the new `PacketListener` classes later in this chapter.

Even without seeing them, you know from this constructor that I'm registering:

- An `OpenStreamHandler` class to handle the special `<stream:stream>` opening tag.
- A `CloseStreamHandler` class to handle the special `</stream:stream>` closing tag.
- A `MessageHandler` class to handle `<message>` Packets.

The `TestThread` will drop all other incoming packets because it does not have any `PacketListeners` to handle them.

The `JabberModel` contains a few member data fields and their access methods. These fields are a convenience for setting up default values throughout the client application. A real client application would have a separate user options object and store these options in configuration files.

The `JabberModel` class data fields and access methods

```
String jabberVersion = "v. 1.0 - ch. 4";
public String getVersion(){ return jabberVersion; }

String sName;
public String getServerName()           {return sName;}
public void   setServerName(String name) {sName = name;}

String sAddress;
public String getServerAddress()         {return sAddress;}
public void   setServerAddress(String addr) {sAddress = addr;}
String sPort;
public String getPort()                  {return sPort;}
public void   setPort(String port)       {sPort = port;}

String user;
public String getUser()                   {return user;}
public void   setUser(String usr)        {user = usr;}

String resource;
public String getResource()               {return resource;}
public void   setResource(String res)    {resource = res;}

public void addStatusListener(StatusListener listener){
    session.addStatusListener(listener);
}

public void removeStatusListener(StatusListener listener){
```

```
        session.removeStatusListener(listener);
    }

    public int getSessionStatus() {
        return session.getStatus();
    }
}
```

There are two interesting features of the code. The first is that there are some convenience methods for registering `StatusListeners` with the `JabberModel`'s `Session` object and obtaining the `Session`'s status. These convenience methods allow us to keep the `Session` object completely encapsulated within the `JabberModel` and prevents any outside classes from directly manipulating the `Session`.

The second thing to note is that we have a separate `serverName` and `serverAddress` field. In normal clients you will only need a server's name (e.g., "shigeoka.com"). The `Socket` class automatically figures out the server's address (e.g., "217.13.31.1") using DNS lookup. However, I tend to develop on isolated development machines and offline laptops. By providing both a server name and a server address, I can have the client act as if it is talking with a server `shigeoka.com` while connecting to a hardcoded address that may not have any real DNS name.

In this case, I can use the loopback address `127.0.0.1` so that I am able to run the client and server on the same machine without any network connection at all.¹⁷ The client and server both think that the server is at `shigeoka.com`, allowing me to use Jabber IDs like `iain@shigeoka.com/work` rather than `iain@127.0.0.1/work`.

Next, the `JabberModel` implements the three remaining tasks that the client model must fulfill: connecting, sending messages, and disconnecting. The most code-intensive is the `connect()` method shown in listing 4.2.

Listing 4.2 The `JabberModel` class `connect` method

```
public void connect(String server,
                   int port,
                   String serverName,
                   String user,
                   String resource)

    throws IOException {
```

¹⁷ The loopback address is a logical network address available to all TCP/IP clients. The address always points to the localhost: the machine you are currently working on. Thus the address provides a virtual loopback connection to yourself.

```

session.setSocket(new Socket(server,port));
session.setStatus(Session.CONNECTED);

(new ProcessThread(packetQueue,session)).start();

String senderJabID = user + "@" + sName + "/" + resource;

Writer out = session.getWriter();
session.setJID(new JabberID(user,sName,resource));
out.write("<?xml version='1.0' encoding='UTF-8' ?>");
out.write("<stream:stream to='");
out.write(sName);
out.write("' from='");
out.write(senderJabID);
out.write("' xmlns='jabber:client' ");
out.write("xmlns:stream='http://etherx.jabber.org/streams'>");
out.flush();
}

```

Create a socket for the session

Session connected as soon as we open the Socket

Start JabberIput Handler in the ProcessThread

Send opening <stream:stream> tag

Connecting to the Jabber server is relatively simple and resembles the creation of a Jabber session in the server. We can even reuse the `ProcessThread` class from the server to parse the incoming Jabber XML and place `Packet` classes into the `PacketQueue`.

Disconnecting and sending messages are supported by even simpler methods as shown in listing 4.3:

Listing 4.3 The `JabberModel` class `disconnect` and `sendMessage` methods

```

public void disconnect() throws IOException {
    session.getWriter().write("</stream:stream> ");
    session.getWriter().flush();
}

public void sendMessage(String recipient,
                        String subject,
                        String thread,
                        String type,
                        String id,
                        String body) throws IOException {

    Packet packet = new Packet("message");

    if (recipient != null){
        packet.setTo(recipient);
    }
    if (id != null){
        packet.setID(id);
    }
}

```

```
}
if (type != null){
    packet.setType(type);
}
if (subject != null){
    packet.getChildren().add(new Packet("subject",subject));
}
if (thread != null){
    packet.getChildren().add(new Packet("thread",thread));
}
if (body != null){
    packet.getChildren().add(new Packet("body",body));
}
packet.writeXML(session.getWriter());
}
}
```

The `disconnect()` method simply sends the closing `</stream>` tag. It does not have to close the socket because the server will automatically close it when it receives the client's closing stream tag.

The `sendMessage()` method creates a `Packet` object and fills it with the information it needs to generate the correct XML for a Jabber `<message>` packet. The `sendMessage()` method is a convenient way of sending Jabber messages from within Java code.

Now that we know which `Packet` classes we must handle, the final step in building the client model is constructing the client packet handler classes. The client packet handling classes straddle the line between the model and view because they must know how to deal with the Jabber `Packet` classes (part of the model) as well as how to update the user interface (part of the view). Our test client has almost no user interface so these classes remain extremely simple.

The client OpenStreamHandler class

The first packet we should react to is the open stream packet that the server sends us.

The client OpenStreamHandler class

```
public class OpenStreamHandler implements PacketListener{

    public void notify(Packet packet){
        Session session = packet.getSession();
        session.setStreamID(packet.getID());
        session.setJID(new JabberID(packet.getTo()));
        session.setStatus(Session.STREAMING);
    }
}
```

Recall that the client initializes the stream so we have already sent the `<stream:stream>` tag to the server. When we receive the server's return opening stream tag, we just need to extract the stream ID from it, and update the session with its new status and information.

Unlike the server, we don't need a `CloseStreamHandler` class in the client. Once we send a closing stream tag using `disconnect()`, the server will close the `Socket` at its earliest opportunity. We will likely never see a closing `</stream:stream>` tag. Notice that if the stream closes without sending a closing `</stream:stream>` tag, the XML document will not be valid and the `SAXparser` will generate a `SAXException`. We must be ready to receive this error and ignore it. (We expect it to happen).

The most important packet handler is the message handler class, which is discussed next.

The Client MessageHandler Class

The client handles all `<message>` packet types::

- Chat
- Normal
- Groupchat
- Headline
- Error

- `Jabber:x:oob` (out-of-band X extensiton)

In clients with a user interface, most of your time will be spent making the display of these messages intuitive and fun for the user. Our simple test client, on the other hand, uses incoming messages for different purposes.

For example, if we wanted to measure the server's message throughput, a test client only needs to count how many messages sent and received during the test. Similarly, a messaging latency test can be conducted by a client sending a message to itself and measuring the time it takes for the message to make its round trip through the server.

In our client, we only need to know that packets are being properly routed to their destination. We can see this by looking at the raw XML passed over the connection. The task can be made easier by simply printing the message when we get a `<message>` packet.

The client MessageHandler class

```
public class MessageHandler implements PacketListener {

    public void notify(Packet packet){
        String type = packet.getType() == null ? "normal" : packet.getType();
        System.out.println("Received " + type + " message: "
            + packet.getChildValue("body"));
        System.out.println("    To: " + packet.getTo());
        System.out.println("    From: " + packet.getFrom());
    }
}
```

We could make the message simpler or more verbose, log the message to a file or database, or myriad other possibilities. For now though, this implementation will meet our needs.

The last step in creating our test client is driving the `JabberModel`. We'll accomplish this with the `TestThread` and `SimpleMessageClient` class.

4.3.4 Using the client model

The `JabberModel` class deals with all the details of Jabber communications. However, by itself, it won't do anything. Like all models in the MVC design pattern, the `JabberModel` is a passive class that reacts to inputs. We must write a class that plays the role of the user as well as the view and controller from the MVC design pattern. The `TestThread` class fulfills this role.

Our `TestThread` class will pull packets from the `PacketQueue` as shown in figure 4.8. However, unlike the `QueueThread`, we know what packets to expect on the `PacketQueue` so we can dispatch the packets with more intelligence. In addition, the `TestThread` is aware of both incoming and outgoing packets. This gives us the opportunity to send and receive packets in an expected order. Deviations from the expected order of outgoing or incoming packets signal a failure of the test. This turns our client programming model into a pseudo blocking method call system rather than the server's event based model.

In blocking systems, you send a packet by calling a method. The method returns with the appropriate response or when the protocol enters its next state. For example, a blocking call to `JabberModel.connect()` would only return when we receive a success result (a failure would throw an exception). The blocking method call system eliminates the entire event-handling model we've been using so far.

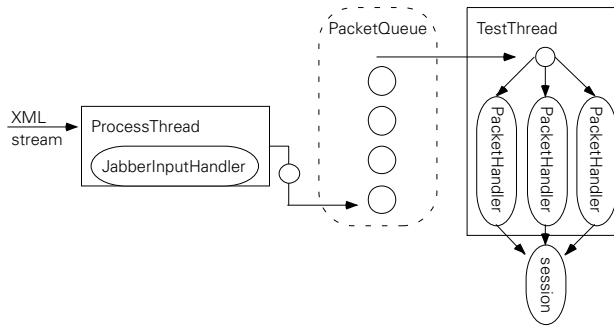


Figure 4.8 The `TestThread` class replaces the `QueueThread`. It intelligently routes packets and triggers tests by sending packets in response to incoming packets and its current state.

It is easier to program networked clients by blocking rather than using events. This is true with Jabber clients as well as those that support common network protocols like NNTP (Usenet news) and POP (email). The tradeoff is relatively straightforward. Event-based systems are easy to make multithreaded, easy to make responsive by running them in multiple threads, and easy to scale by using multiple processors or multiple machines to handle events.

All of these factors make event-based systems ideal for servers. Most of the Jabber server's interactions only require "local" information that is often present in the packet being processed or contained in the session context. You can design most of its actions as a simple, one-packet response.

Clients, on the other hand, usually use protocols in long, complex sessions where the context of events is just as important as the event itself.¹⁸ For example, a client may wish to send a message. This simple goal ends up requiring a number of steps that must be carried out in the correct order. We know that we can't send a message without being authenticated. In addition, we can be authenticated without having established a Jabber XML stream with the server and having an account on the server. Finally, we can't establish a Jabber XML stream without being connected with the server.

¹⁸ It can be complex enough that it may become worth the effort to use state-machines to carry out client tasks. In fact, I would recommend that designers of fully featured clients invest the effort to write or purchase a state-machine framework. Your client code will become easier to manage. In addition, it provides the groundwork for easy extensibility via scripting languages or plug-ins and allows you to automate Jabber clients. State-machines are a standard computer science model of computing systems. To learn more about state-machines, consult an introductory computer science textbook.

The client has little need for multitasking within its packet handling system. It will typically participate in one conversation at a time. Finally, clients usually don't need to be designed to be scalable. They are almost always limited by the abilities of the user to create input events and understand incoming information.

Our protocol tests will be carried out in a simple blocking style by sending a packet and waiting for the correct reply packet. This is similar to the approach we take when interacting with the server using telnet:

- 1 Send a packet.
- 2 Wait for a result.
- 3 See if it matches what we expect.
- 4 If so, go to the next step in the conversation.

For our first version, the client connects, sends a message, and disconnects. We need to carry out the two sides of the conversations in different threads so they can occur in parallel. Hence, the `TestThread` is a `java.lang.Thread` child class to allow more than one to run simultaneously.

The `TestThread` class is actually a base class of the test-specific classes we'll use in each chapter. It provides the basic packet-handling features we've seen in the `QueueThread`. In addition, we'll add a helper method that will allow the `TestThread` subclasses to wait for packets.

The `TestThread` class (listing 4.4) begins just like the `QueueThread`. The sole exception is the empty `run()` method and a simple way of assigning a `JabberModel` to the `TestThread` using the `setModel()` method. Subclasses will override the `run()` method to provide test specific code.

Listing 4.4 The `TestThread` class packet handling code

```
public class TestThread extends Thread {

    public void run(){
    }

    JabberModel model;
    public void setModel(JabberModel newModel){
        model = newModel;
    }

    PacketQueue packetQueue = new PacketQueue();
    public PacketQueue getQueue() { return packetQueue; }

    HashMap packetListeners = new HashMap();
```

```

public boolean addListener(PacketListener listener, String element){
    if (listener == null || element == null){
        return false;
    }
    packetListeners.put(listener,element);
    return true;
}

public boolean removeListener(PacketListener listener){
    packetListeners.remove(listener);
    return true;
}

```

The packet listener management code is essentially the same as that in the `QueueThread`.¹⁹ We can copy the code from the `QueueThread`'s `run()` method into a `notifyHandler()` method to provide the same treatment of packet handlers in the `TestThread`.

The `TestThread` class `notifyHandler` method sends packets to registered handlers

```

void notifyHandlers(Packet packet){
    try {
        Packet child;
        String matchString = packet.getElement();

        synchronized(packetListeners){
            Iterator iter = packetListeners.keySet().iterator();
            while (iter.hasNext()){
                PacketListener listener = (PacketListener)iter.next();
                String listenerString = (String)packetListeners.get(listener);
                if (listenerString.equals(matchString)){
                    listener.notify(packet);
                }
            }
        }
    } catch (Exception ex){
        Log.error("TestThread: ", ex);
    }
}

```

As mentioned earlier, subclasses will conduct tests by sending packets out, and then wait for the correct packet to arrive. We can create a convenience method that makes waiting for specific packets simpler:

¹⁹ The similarities suggest a good opportunity for creating a common base class for `TestThread` and `QueueThread`.

The TestThread class waitFor method waits for the "correct" packet

```

Packet waitFor(String element, String type){
    for( Packet packet = packetQueue.pull();
        packet != null;
        packet = packetQueue.pull()) {
        notifyHandlers(packet);
        if (packet.getElement().equals(element)){
            if (type != null){
                if (packet.getType().equals(type)){
                    return packet;
                }
            } else {
                return packet;
            }
        }
    }
    return null;
}

```

The `waitFor()` method uses the packet's element name, and optionally a particular type attribute to filter out the packet we're looking for. All other packets are sent to their packet handlers using the `notifyHandlers()` method. The `waitFor()` method returns the first matching packet it finds.

We'll create two `TestThread` subclasses as inner classes of the `SimpleMessageClient` class shown in listing 4.5. The `SimpleMessageClient` class is a main application class that can be launched as a Java application. We provide the standard `main()` method for that purpose. The constructor takes care of most of the work, setting up the two test threads and models.

Listing 4.5 The SimpleMessageClient class

```

class SimpleMessageClient {

    public static void main(String[] args){
        Client client = new Client();
    }

    public SimpleMessageClient(){
        String server = System.getProperty("jab.server.name", "localhost");
        String address = System.getProperty("jab.server.address", "127.0.0.1");
        String port = System.getProperty("jab.server.port", "5222");

        BuffyTestThread buffyTT = new BuffyTestThread();
        JabberModel buffyModel = new JabberModel(iainTT);
        AngelTestThread angelTT = new AngelTestThread();
        JabberModel angelModel = new JabberModel(angelTT);
    }
}

```

**The client application
simply creates Client
object**

**Extract
settings from
system
properties**

```

buffyModel.setServerName(server);
buffyModel.setServerAddress(address);
buffyModel.setPort(port);

buffyModel.setUser("buffy");
buffyModel.setResource("dev");

angelModel.setServerName(server);
angelModel.setServerAddress(address);
angelModel.setPort(port);

angelModel.setUser("angel");
angelModel.setResource("dev");

buffyTT.setModel(buffyModel);
buffyTT.start(); ←
angelTT.setModel(angelModel);
angelTT.start(); ←
}

```

Start two test threads

The actual tests are conducted in the two inner classes, `BuffyTestThread` and `AngelTestThread`, shown in listings 4.6 and 4.7. We'll simulate a short conversation between "buffy" and "angel." Buffy will start by sending a message and waiting for a reply. Angel will receive Buffy's message and reply. Angel doesn't need to wait for a reply so the `AngelTestThread` will disconnect as soon as it sends the reply message.

Listing 4.6 The `BuffyTestThread` inner class

```

public class BuffyTestThread extends TestThread {

    public void run(){
        try {
            model.connect();
            waitFor("stream:stream",null);
            model.sendMessage("angel@" + model.getServerName(),
                "Want to patrol?",
                "thread_id",
                "normal",
                "msg_id_buffy",
                "Hey, do you wanted to patrol with me tonight?");
            waitFor("message",null);
            model.disconnect();
        }
    }
}

```

```
    } catch (Exception ex){  
        ex.printStackTrace();  
    }  
}
```

Our blocking style of programming is well-suited to client interactions where we typically need to do one thing at a time. This is in contrast to the event-based model used by the server's `QueueThread` where we expect many things to be happening in parallel. The `AngelTestThread` provides the other side of the Jabber conversation.

Listing 4.7 The `AngelTestThread` inner class

```
public class AngelTestThread extends TestThread {  
  
    public void run(){  
        try {  
            model.connect();  
            for (Packet packet = waitFor("message",null);  
                packet.getFrom().startsWith("buffy");  
                packet = waitFor("message",null)){  
            }  
            model.sendMessage("buffy@" + model.getServerName(),  
                              "Re: Want to patrol?",  
                              "thread_id",  
                              "normal",  
                              "msg_id_angel",  
                              "Sure, I'd love to go.");  
            model.disconnect();  
        } catch (Exception ex){  
            ex.printStackTrace();  
        }  
    }  
}
```

Notice that the `AngelTestThread` uses an empty for loop to ensure that it waits for a message from “buffy” before sending a reply. Many Jabber servers send a welcome message to clients when they log on so we want to make sure we don't react to that. Of course, a normal Jabber server will require you to authenticate and indicate that you are available to receive messages before any are sent. However,

our client is testing our server which lacks these features so we can skip these authentication steps until we add these features in future chapters.

EXPLOITING THE STATUS EVENT MODEL

Every GUI application is riddled with little bits of minutiae that can drive you crazy. One of them is the need to maintain a consistent application state at all times. In large applications, code scattered all over your application can change the state of your application at any time. The rest of the application classes, and most importantly the GUI, must be updated appropriately.

Session status is just one of many situations where we must continuously maintain a consistent application state. In this situation, a user-friendly application should be updating the appearance and the enabled status of menu items, buttons, and windows according to the `Session`'s status. We can use the status event notification feature we designed in the `Session` class to help automate the state update process.

As your application grows, you will be forced to decide whether to perform all status updates in one `StatusListener` class or to split the role among many smaller `StatusListener` implementation classes. Typically code will naturally evolve as a single large `StatusListener` class. Unfortunately, throwing “everything but the kitchen sink” into one class is usually not a good idea.

To eliminate the need for one `StatusListener` class, a Swing-based GUI client might use a specialized `javax.swing.JButton` that implements the `StatusListener` interface. The code might look like this:

```
class StreamEnabledButton extends JButton
    implements StatusListener{

    public void notify(int status){

        switch(status){

            case Session.DISCONNECTED:
                setEnabled(false);
                break;

            case Session.STREAMING:
                setEnabled(true);
                break;

        }
    }
}
```

You can then add the `StreamEnabledButton` as a `StatusListener` to the `JabberModel` and it will automatically enable and disable itself when appropriate.

All buttons that you want to enable when the `Session` object is in the streaming state can use this class rather than `JButton`.

In addition, there are many Jabber features that Jabber servers will not allow unless you are authenticated.²⁰ These features can use a similar customized `StatusListener` implementation and a new `Session` status for `AUTHENTICATED` to automatically enable and disable these features as well.

4.3.8 Results

Does it seem possible that in two short chapters we have created a Jabber client and server that can support basic IM? Well, try it out for yourself. Start the server from chapter 3, and then launch our Jabber client.

The `SimpleMessageClient` uses `java.lang.System` properties for many of its settings. You can set these properties on the command line using JVM options. For standard JVMs the `-D` option is used to pass these values. For example, I want to start the client so that it uses the server address `10.0.0.5`:

```
java -Djab.server.address=10.0.0.5 SimpleMessageClient
```

A shell script or batch file can reduce the amount of typing you need to do, and help to launch the clients sequentially or simultaneously. Does the client send messages to itself as we would expect?

Try creating two `BuffyTestThreads` in the client with the different resource names. Where does Angel's reply message go? Messages should be delivered on a first-come, first-served basis. Are they? Is the server's message delivery behavior consistent with the server packet routing behavior we implemented in chapter 3? If you have two or even three computers on a network, it is even more impressive to create separate client applications that you can run on separate machines.

Congratulations! You're Jabbering!

NOTE Client Is a Work in Progress Be careful using the client "as-is". It lacks some critical "spit and polish" to make it safe for heavy use. For example, try sending a message with the '`<`' or '`>`' characters in the message body. It will crash the parser. Our client does not "normalize" the text before placing it into the `Packet`. Normalizing XML text changes the '`<`' character to the string "`<`," which is then safe to transport inside of XML character data.

²⁰ Authentication is covered in chapter 7.

4.4 *Conclusions*

The simplicity of the Jabber protocols and the power of Java have allowed us to create a miniature IM system in only two chapters. Playing with the current client and server will reveal how powerful even this simple Jabber system is. In fact, just a little user interface customization can make this system perfect for a lightweight communication system in a small home or office LAN.

The crucial IM feature missing from the software is support for presence. Presence lets us know who is online and if they are willing to communicate with us. This critical protocol enables features like chat to meet their full potential and is essential before we can conduct groupchat conferences. We'll dive into the Jabber presence protocols, and add groupchat conference support to the server and client in the next chapter.