Steal This Book!

Yes, you read that right. Steal this book. For free.

Nothing. Zero. Zilch. Nadda. Zip.

Undoubtedly you're asking yourself, "Why would he give away a book he probably spent six grueling months writing? Is he crazy?"

The answer…is yes. Sort of. I know that every day you're faced with hundreds of computer titles, and a lot of them don't deliver the value or the information you need. So here's the deal: I'll give you this book (or this chapter, if you've only downloaded part of the book) for free provided you do me a couple favors:

1. **Send this book to your friends:** No, not your manager. Your "with it" computer friends who are looking for the next Big Thing. JXTA is it. Trust me. They want to know about it.

2. **Send a link to the book's web site:** Maybe the book is too big to send. After all, not everyone can have a fibre optic Internet connection installed in their bedroom. The site, at **www.brendonwilson.com/projects/jxta**, provides chapter-sized PDFs for easy downloading by the bandwidth-challenged.

3. **Visit the book's web site:** Being a professional developer, you probably have Carpal Tunnel Syndrome and shudder at the idea of typing in example source code. Save yourself the trouble. Go to **www.brendonwilson.com/projects/jxta** and download the source code. And while you're there, why not download some of the chapters you're missing?

4. **Buy the book:** You knew there had to be a catch. Sure, the book's PDFs are free, but I'm hoping that enough of you like the book so much that you have to buy a copy. Either that, or none of you can stand to read the book from a screen (or, worse yet, print it all out <shudder>) and resort to paper to save what's left of your eyesight. The book is available at your local bookstore or from **Amazon.com** (at a **handsome discount**, I might add).

I now return to your regularly scheduled program: enjoy the book!

11

A Complete Sample Application

At this point in the book, you should be familiar with all the pieces involved in a JXTA solution, but not necessarily how to put them together into a complete application. This chapter provides a complete sample application that illustrates how to assemble the pieces provided by the JXTA reference implementation into a complete P2P solution.

The sample application in this chapter demonstrates the creation of a JXTA-based chat application with simple presence management. The application is similar to other popular instant-messaging clients, but it incorporates fewer features. The sample application incorporates many of the JXTA protocols to provide a complete solution.

The main features of this application are the capability to chat with a remote user and monitor remote users' presence status. This status allows a user to determine whether one of his contacts (or "buddies") is currently online, offline, busy, or temporarily away from the computer.

Creating the Presence Service

The Presence service provides a mechanism for exchanging presence information with another user. For this application, the Presence service is fairly unsophisticated and doesn't attempt to address more complex presence-management problems. For example, this Presence service assumes that a user is on the network at only a single location using a single peer. In addition, the Presence service assumes that JXTA provides a reliable transport, which is not always a good assumption. Although the reference implementation uses TCP, a reliable transport, there are no guarantees on a given JXTA peer that the peer will be using a reliable transport.

The JXTA Community is currently working on a fully featured Presence Management Framework that will provide much more functionality than this sample application's Presence service. For more information on the Presence Management Framework, see the project web site at `presence.jxta.org`. This example Presence service is simply an example designed to show how the various pieces explored over the course of this book fit together into a single application.

At first glance, it might appear that the Presence service should be built using the Resolver service. However, building the Presence service using the Resolver service would require a peer to send a query to a remote peer every time that it required presence information. The network overhead incurred by this technique would be undesirable.

It would be better if presence information could be published using the Discovery service, thus allowing presence information to be cached by other peers. Of course, the risk here is that the presence information might be stale, but this can be resolved by publishing the advertisement with a short lifetime. Another disadvantage is that this method does not scale well to large numbers of peers. This is something that is acceptable for this simple application, but it would not be acceptable in a large-scale P2P solution.

To implement the Presence service, three components are needed:

- **An advertisement**—The Presence service needs a format for the presence information to be exchanged with other peers using the Discovery service. The formatting and parsing logic for the advertisement must be implemented to allow the Presence service to handle the advertisement in an encapsulated fashion.
- **A service**—The Presence service itself needs to provide an interface that third-party developers can use to interact with the service. An implementation of the service's interface is required to handle the details of using the Discovery service to publish and discover presence information.

- **A listener**—This application needs some way of receiving notification that new presence information has been received by the Presence service. A listener interface that can be implemented and registered with the Presence service solves this problem.

The creation of these elements is the subject of the next two sections.

The Presence Advertisement

The Presence Advertisement is responsible for describing the current presence status of a particular user. To uniquely identify both a peer and the user, the Presence Advertisement needs the Peer ID, plus one other piece of identification that is unique to the user. For this purpose, the Presence Advertisement uses the user's email address to uniquely identify the user.

To represent the presence information, the Presence Advertisement uses the XML format shown in Listing 11.1.

Listing 11.1 **The Presence Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<PresenceAdvertisement>
  <PeerID> . . . </PeerID>
  <EmailAddress> . . . </EmailAddress>
  <PresenceStatus> . . . </PresenceStatus>
  <Name> . . . </Name>
</PresenceAdvertisement>
```

The content of the Presence Advertisement describes all the elements related to a user's presence on the P2P network:

- **`PeerID`**—A required element containing the Peer ID identifying the peer that the user is currently using on the network.
- **`EmailAddress`**—A required element containing the user's email address. The email address is used as a unique identifier for a user.
- **`PresenceStatus`**—A required element containing an integer representing the user's presence status. A value of `0` indicates that the user is offline, `1` indicates that the user is online, `2` indicates that the user is currently busy, and `3` indicates that the user is temporarily away from the computer.
- **`Name`**—An optional element containing a display name or common name for the user.

To implement the Presence Advertisement, you define an abstract class derived from the `net.jxta.document.Advertisement` class. This class, `PresenceAdvertisement`, is shown in Listing 11.2.

Listing 11.2 **Source Code for *PresenceAdvertisement.java***

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Advertisement;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;

import net.jxta.id.ID;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.presence.PresenceService;


/**
 * An abstract class defining an advertisement containing the elements used
 * to describe a user's presence status. A user is assumed to be uniquely
 * described by his or her email address.
 */
public abstract class PresenceAdvertisement extends Advertisement
{
    /**
     * The root element for the advertisement's XML document.
     */
    private static final String advertisementType = "PresenceAdvertisement";

    /**
     * The email address identifying the user whose presence information
     * this advertisement describes.
     */
    private String emailAddress = null;

    /**
     * A simple name for the user specified by the advertisement's
     * email address.
     */
    private String name = null;

    /**
     * The Peer ID locating the peer on the network.
     */
    private String peerID = null;
```

```
/**
 * A simple descriptor identifying the user's presence status.
 * The user can indicate that he or she is online, offline, busy, or
 * away.
 */
private int presenceStatus = PresenceService.OFFLINE;


/**
 * Returns the advertisement type for the advertisement's document.
 *
 * @return  the advertisement type String.
 */
public static String getAdvertisementType()
{
    return advertisementType;
}

/**
 * Returns the email address String describing the user whose presence
 * status is described by this advertisement.
 *
 * @return  the email address for the advertisement.
 */
public String getEmailAddress()
{
    return emailAddress;
}

/**
 * Returns a unique identifier for this document. There is none for
 * this advertisement type, so this method returns the null ID.
 *
 * @return  the null ID.
 */
public ID getID()
{
    return ID.nullID;
}

/**
 * Returns the simple name for the user described by this advertisement.
```

continues

Listing 11.2 **Continued**

```
     *
     * @return  the user's name.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Returns the Peer ID of the user described by this advertisement.
     *
     * @return  the Peer ID of the user.
     */
    public String getPeerID()
    {
        return peerID;
    }

    /**
     * Returns the presence status information of the user described by
     * this advertisement.
     *
     * @return  the user's status information.
     */
    public int getPresenceStatus()
    {
        return presenceStatus;
    }

    /**
     * Sets the email address String describing the user whose presence
     * status is described by this advertisement.
     *
     * @param   emailAddress the email address for the advertisement.
     */
    public void setEmailAddress(String emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    /**
```

```
     * Sets the simple name for the user described by this advertisement.
     *
     * @param   name the user's name.
     */
    public void setName(String name)
    {
        this.name = name;
    }

    /**
     * Sets the Peer ID identifying the peer's location on the P2P network.
     *
     * @param   peerID the Peer ID for the advertisement.
     */
    public void setPeerID(String peerID)
    {
        this.peerID = peerID;
    }

    /**
     * Sets the presence status information of the user described by this
     * advertisement.
     *
     * @param   presenceStatus the user's status information.
     */
    public void setPresenceStatus(int presenceStatus)
    {
        this.presenceStatus = presenceStatus;
    }
}
```

The `PresenceAdvertisement` class defines basic accessors to set and retrieve the advertisement's various parameters. In addition, the class defines the static `getAdvertisementType` method to return the root element tag used by the Presence Advertisement.

`PresenceAdvertisement` also defines the `getID` method that is used by the Cache Manager to index the advertisement in the cache. The `ID` returned by `getID` should uniquely identify the advertisement. To avoid having to implement your own `ID` implementation, `PresenceAdvertisement` returns `ID.nullID`. This null `ID` prompts the Cache Manager to use a hash of the advertisement to index the advertisement in the cache, and it is sufficient for your purposes.

The `Advertisement.getDocument` method is not defined by `PresenceAdvertisement`, to allow the implementation of `PresenceAdvertisement` to define logic for parsing and formatting a Presence Advertisement. This method is implemented by the `PresenceAdv` subclass, shown in Listing 11.3, using the JXTA reference implementation.

Listing 11.3 **Source Code for *PresenceAdv.java***

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * An implementation of the PresenceAdvertisement abstract class. This
 * class is responsible for parsing and formatting the XML document
 * used to define presence information for a peer.
 */
public class PresenceAdv extends PresenceAdvertisement
{
    /**
     * A convenient constant for the XML MIME type.
     */
    private static final String mimeType = "text/xml";
```

```
/**
 * The element name for the presence advertisement's email address info.
 */
private static final String tagEmailAddress = "EmailAddress";

/**
 * The element name for the presence advertisement's simple name info.
 */
private static final String tagName = "Name";

/**
 * The element name for the presence advertisement's Peer ID.
 */
private static final String tagPeerID = "PeerID";

/**
 * The element name for the presence advertisement's status info.
 */
private static final String tagPresenceStatus = "PresenceStatus";


/**
 * An Instantiator used by the AdvertisementFactory to instantiate
 * this class in an abstract fashion.
 */
public static class Instantiator
    implements AdvertisementFactory.Instantiator
{
    /**
     * Returns the identifying type of this advertisement.
     *
     * @return  the name of the advertisement's root element.
     */
    public String getAdvertisementType()
    {
        return PresenceAdvertisement.getAdvertisementType();
    }

    /**
     * Returns a new PresenceAdvertisement implementation instance.
     *
     * @return  a new presence advertisement instance.
     */
```

continues

Listing 11.3 **Continued**

```
    public Advertisement newInstance()
    {
        return new PresenceAdv();
    }

    /**
     * Instantiates a new PresenceAdvertisement implementation instance
     * populated from the given root element.
     *
     * @param   root the root of the object tree to use to populate the
     *           advertisement object.
     * @return  a new populated presence advertisement instance.
     */
    public Advertisement newInstance(Element root)
    {
        return new PresenceAdv(root);
    }
};

/**
 * Creates a new presence advertisement.
 */
public PresenceAdv()
{
    super();
}

/**
 * Creates a new presence advertisement by parsing the given stream.
 *
 * @param       stream the InputStream source of the advertisement data.
 * @exception   IOException if the advertisement can't be parsed from
 *              the stream.
 */
public PresenceAdv(InputStream stream) throws IOException
{
    super();

    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);
```

```
    readAdvertisement(document);
}

/**
 * Creates a new presence advertisement by parsing the given document.
 *
 * @param   document the source of the advertisement data.
 */
public PresenceAdv(Element document) throws IllegalArgumentException
{
    super();

    readAdvertisement((TextElement) document);
}

/**
 * Returns a Document object containing the advertisement's
 * document tree.
 *
 * @param        asMimeType the desired MIME type for the
 *               advertisement rendering.
 * @return       the Document containing the advertisement's document
 *               object tree.
 * @exception    IllegalArgumentException thrown if either the email
 *               address or the Peer ID is null.
 */
public Document getDocument(MimeMediaType asMimeType)
    throws IllegalArgumentException
{
    // Check that the required elements are present.
    if ((null != getEmailAddress()) && (null != getPeerID()))
    {
        PipeAdvertisement pipeAdv = null;

        StructuredDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                asMimeType, getAdvertisementType());
        Element element;

        // Add the Peer ID information.
        element = document.createElement(tagPeerID, getPeerID());
        document.appendChild(element);
```

continues

Listing 11.3 **Continued**

```
            // Add the email address information.
            element = document.createElement(
                tagEmailAddress, getEmailAddress());
            document.appendChild(element);

            // Add the display name information, if any.
            if (null != getName())
            {
                element = document.createElement(tagName, getName());
                document.appendChild(element);
            }

            // Add the presence status information.
            element = document.createElement(tagPresenceStatus,
                Integer.toString(getPresenceStatus()));
            document.appendChild(element);

            return document;
        }
        else
        {
            throw new IllegalArgumentException(
                "Missing email address or peer ID!");
        }
    }

    /**
     * Parses the given document tree for the presence advertisement.
     *
     * @param       document the object containing the presence
     *              advertisement data.
     * @exception   IllegalArgumentException if the document is not a
     *              presence advertisement, as expected.
     */
    public void readAdvertisement(TextElement document)
        throws IllegalArgumentException
    {
        if (document.getName().equals(getAdvertisementType()))
        {
            Enumeration elements = document.getChildren();
```

```
            while (elements.hasMoreElements())
            {
                TextElement element = (TextElement) elements.nextElement();

                // Check for the email address element.
                if (element.getName().equals(tagEmailAddress))
                {
                    setEmailAddress(element.getTextValue());
                    continue;
                }

                // Check for the display name element.
                if (element.getName().equals(tagName))
                {
                    setName(element.getTextValue());
                    continue;
                }

                // Check for the email address element.
                if (element.getName().equals(tagPresenceStatus))
                {
                    setPresenceStatus(
                        Integer.parseInt(element.getTextValue()));
                    continue;
                }

                // Check for the Peer ID element.
                if (element.getName().equals(tagPeerID))
                {
                    setPeerID(element.getTextValue());
                    continue;
                }
            }
        }
        else
        {
            throw new IllegalArgumentException(
                "Not a PresenceAdvertisement document!");
        }
    }

    /**
```

continues

Listing 11.3 **Continued**

```
     * Returns an XML String representation of the advertisement.
     *
     * @return  the XML String representing this advertisement.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
                getDocument(new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

In addition to providing a `getDocument` implementation, the `PresenceAdv` class provides several constructors that provide advertisement parsing functionality. All the parsing and formatting functionality is built using the `net.jxta.document` classes to handle manipulating the XML object tree.

In Chapter 4, "The Peer Discovery Protocol," you learned about the `AdvertisementFactory` class and how it could be used to instantiate an `Advertisement` implementation in an abstract manner using a `String`. Usually, this `String` comes from the abstract advertisement implementation class's `getAdvertisementType` method:

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            PeerAdvertisement.getAdvertisementType());
```

For `AdvertisementFactory` to be capable of doing the same with the Presence Advertisement implementation, the implementation class must be registered with `AdvertisementFactory`. To register an implementation class, the application will need to call `AdvertisementFactory.registerAdvertisementInstance`:

```
public static boolean registerAdvertisementInstance(
    String rootType, Instantiator instantiator)
```

The `rootType String` defines the advertisement type `String` that will be mapped to the advertisement implementation. The `instantiator` parameter is an instance of an implementation of the `AdvertisementFactory.Instantiator` class. The `PresenceAdv` class shown in Listing 11.3 provides an implementation of this class that the `AdvertisementFactory` can use to create a new `PresenceAdv` instance. To register the `PresenceAdv` implementation with `AdvertisementFactory`, the sample application must execute the following:

```
AdvertisementFactory.registerAdvertisementInstance(
    PresenceAdvertisement.getAdvertisementType(),
    new PresenceAdv.Instantiator());
```

This call needs to be executed when the application starts, before any other class attempts to use `AdvertisementFactory` to instantiate a Presence Advertisement. In the example, `registerAdvertisementInstance` is called in the Presence service's `init` method to ensure that the advertisement type is registered.

The Presence Service Definition

Although this application could handle publishing and discovering Presence Advertisements directly, it would be better if a developer could avoid handling the Presence Advertisement and interacting with the Discovery service. If you define an interface for the Presence service, the solution will be more flexible and developers will be shielded from future implementation changes.

For example, if you didn't define an interface, a developer would have to use the Presence Advertisement classes and the Discovery service directly. What happens if the developer decides later that the application needs to use a mechanism other than the Discovery service for distributing and discovering Presence Advertisements? The application's code will probably need to be changed in several places. However, if the basic functionality of distributing and discovering presence information is defined as an interface, the developer can simply provide a different implementation of the interface and change the implementation that is used by the application.

The interface for the Presence service must allow a developer to do only two things: announce presence information and find presence information. Part of finding presence information involves notifying listener objects when presence information is found or received, necessitating some way of registering and unregistering listeners. All this functionality is defined by the `PresenceService` interface shown in Listing 11.4.

Listing 11.4 **Source Code for *PresenceService.java***

```
package com.newriders.jxta.chapter11.presence;

import net.jxta.service.Service;


/**
 * An interface for the Presence service, a service that allows peers to
 * exchange presence status information specifying their current status
 * (offline, online, busy, away). This interface defines the operations
 * that a developer can expect to use to manipulate the Presence service,
 * regardless of which underlying implementation of the service is being
 * used.
 */
public interface PresenceService extends Service
{
    /**
     * The module class ID for the Presence class of service.
     */
    public static final String refModuleClassID =
        "urn:jxta:uuid-59A9A948905341119EAB8630EED42AB905";

    /**
     * A status value indicating that a user is currently online but
     * is temporarily away from the device.
     */
    public static final int AWAY= 3;

    /**
     * A status value indicating that a user is currently online but
     * is busy and does not want to be disturbed.
     */
    public static final int BUSY = 2;

    /**
     * A status value indicating that a user is currently offline.
     */
    public static final int OFFLINE = 0;

    /**
     * A status value indicating that a user is currently online.
     */
```

```
    public static final int ONLINE = 1;


    /**
     * Add a listener object to the service. When a new Presence Response
     * Message arrives, the service will notify each registered listener.
     *
     * @param   listener the listener object to register with the service.
     */
    public void addListener(PresenceListener listener);

    /**
     * Announce updated presence information within the peer group.
     *
     * @param   presenceStatus the updated status for the user identified
     *          by the email address.
     * @param   emailAddress the email address used to identify the user
     *          associated with the presence info.
     * @param   name a display name for the user associated with the
     *          presence info.
     */
    public void announcePresence(int presenceStatus, String emailAddress,
        String name);

    /**
     * Sends a query to find presence information for the user specified
     * by the given email address. Any response received by the service
     * will be dispatched to registered PresenceListener objects.
     *
     * @param   emailAddress the email address to use to find presence info.
     */
    public void findPresence(String emailAddress);

    /**
     * Removes a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new Presence Response
     * Message arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public boolean removeListener(PresenceListener listener);
}
```

To allow developers to handle notification of newly received presence information, the `PresenceService` class enables a developer to register and unregister a listener using the `addListener` and `removeListener` methods. The `PresenceListener` interface used by both of these methods is shown in Listing 11.5.

Listing 11.5 **Source Code for *PresenceListener.java***

```
package com.newriders.jxta.chapter11.presence;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * An interface to encapsulate an object that listens for notification
 * from the PresenceService of newly arrived presence information.
 */
public interface PresenceListener
{
    /**
     * Notify the listener of newly arrived presence information.
     *
     * @param   presenceInfo the newly received presence information.
     */
    public void presenceUpdated(PresenceAdvertisement presenceInfo);
}
```

The implementation of `PresenceService` that will be used by the sample application relies on the `DiscoveryService` to handle publishing and discovering Presence Advertisements. The `PresenceService` implementation is shown in Listing 11.6.

Listing 11.6 **Source Code for *PresenceServiceImpl.java***

```
package com.newriders.jxta.chapter11.impl.presence;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Enumeration;
import java.util.Vector;

import net.jxta.discovery.DiscoveryEvent;
```

```
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;

import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.ID;

import net.jxta.impl.protocol.DiscoveryResponse;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.ModuleImplAdvertisement;

import net.jxta.service.Service;

import com.newriders.jxta.chapter11.impl.protocol.PresenceAdv;

import com.newriders.jxta.chapter11.presence.PresenceListener;
import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * The implementation of the PresenceService interface. This service
 * builds on top of the Discovery service to provide the functionality
 * for requesting and providing presence information.
 */
public class PresenceServiceImpl implements PresenceService,
    DiscoveryListener
{
    /**
     * The Module Specification ID for the Presence service.
     */
    public static final String refModuleSpecID =
        "urn:jxta:uuid-59A9A948905341119EAB8630EED42AB"
        + "9F4611FF6377C4931AE71BE299B9F34DF06";
```

continues

Listing 11.6 **Continued**

```
/**
 * The default expiration timeout for published presence advertisements.
 * Set to 1 minute.
 */
private static final int DEFAULT_EXPIRATION = 1000 * 60 * 1;

/**
 * The default lifetime for published presence advertisements.
 * Set to 1 minutes.
 */
private static final int DEFAULT_LIFETIME = 1000 * 60 * 5;

/**
 * The element name for the presence advertisement's email address info.
 */
private static final String tagEmailAddress = "EmailAddress";

/**
 * The Discovery service used to publish presence information.
 */
private DiscoveryService discovery = null;

/**
 * The Module Implementation advertisement for this service.
 */
private Advertisement implAdvertisement = null;

/**
 * The local Peer ID.
 */
private String localPeerID = null;

/**
 * The peer group to which the service belongs.
 */
private PeerGroup peerGroup = null;

/**
 * A unique query ID that can be used to track a query.
 */
private int queryID = 0;
```

```
/**
 * The set of listener objects registered with the service.
 */
private Vector registeredListeners = new Vector();


/**
 * PresenceServiceImpl constructor comment.
 */
public PresenceServiceImpl()
{
    super();
}

/**
 * Add a listener object to the service. When a new Presence Response
 * Message arrives, the service will notify each registered listener.
 * This method is synchronized to prevent multiple threads from
 * altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
public synchronized void addListener(PresenceListener listener)
{
    registeredListeners.addElement(listener);
}

/**
 * Announce presence status information to the peer group.
 *
 * @param   presenceStatus the current status to announce.
 * @param   emailAddress the user's email address.
 * @param   name the user's display name.
 */
public void announcePresence(int presenceStatus, String emailAddress,
    String name)
{
    if (discovery != null)
    {
        /*
        PresenceAdvertisement presenceInfo = (PresenceAdvertisement)
            AdvertisementFactory.newAdvertisement(
```

continues

Listing 11.6 **Continued**

```
                PresenceAdvertisement.getAdvertisementType());
        */
        // In some earlier versions of JXTA, registering the
        // advertisement doesn't work properly. To work around this,
        // simply instantiate the advertisement implementation directly.
        // This is not the recommended way to get an advertisement.
        // The recommended way is shown in the commented line
        // preceeding this comment.
        PresenceAdvertisement presenceInfo = new PresenceAdv();

        // Configure the new advertisement.
        presenceInfo.setPresenceStatus(presenceStatus);
        presenceInfo.setEmailAddress(emailAddress);
        presenceInfo.setName(name);
        presenceInfo.setPeerID(localPeerID);

        try
        {
            // Publish the advertisement locally.
            discovery.publish(presenceInfo, DiscoveryService.ADV,
                DEFAULT_EXPIRATION, DEFAULT_LIFETIME);
        }
        catch (IOException e)
        {
            System.out.println("Error publishing locally: " + e);
        }

        // Publish the advertisement remotely.
        discovery.remotePublish(presenceInfo, DiscoveryService.ADV,
            DEFAULT_LIFETIME);
    }
}

/**
 * Handle notification of arriving discovery response messages,
 * determine whether the response contains presence information,
 * and, if so, dispatch the presence information to registered
 * PresenceListeners.
 *
 * @param   event the object containing the discovery response.
 */
```

```
public void discoveryEvent(DiscoveryEvent event)
{
    DiscoveryResponseMsg response = event.getResponse();

    // Extract the PresenceAdvertisement from the response.
    Enumeration responses = response.getResponses();
    while (responses.hasMoreElements())
    {
        String responseElement = (String) responses.nextElement();

        // Check for null response advertisement.
        if (null != responseElement)
        {
            // Parse the advertisement.
            try
            {
                ByteArrayInputStream stream =
                    new ByteArrayInputStream(
                        responseElement.getBytes());

                /*
                PresenceAdvertisement advertisement =
                    (PresenceAdvertisement)
                        AdvertisementFactory.newAdvertisement(
                            new MimeMediaType("text/xml"), stream);
                */

                // In some earlier versions of JXTA, registering the
                // advertisement doesn't work properly. To work around
                // this, simply instantiate the advertisement
                // implementation directly. This is not the recommended
                // way to get an advertisement. The recommended way
                // is shown in the commented line preceeding this
                // comment.

                PresenceAdvertisement advertisement =
                    new PresenceAdv(stream);

               // Dispatch the advertisement to the registered presence
                // listeners.
                Enumeration listeners = registeredListeners.elements();
                while (listeners.hasMoreElements())
                {
```

continues

Listing 11.6 **Continued**

```
                        PresenceListener listener =
                            (PresenceListener) listeners.nextElement();

                        // Notify the listener of the presence update.
                        listener.presenceUpdated(advertisement);
                    }
                }
                catch (IOException e)
                {
                    // Obviously not a response to our query for presence
                    // information. Ignore the error.
                    System.out.println("Error in discoveryEvent: " + e);
                }
            }
            else
            {
                System.out.println("Response advertisement is null!");
            }
        }
    }

    /**
     * Sends a query to find presence information for the user specified
     * by the given email address. Any response received by the service
     * will be dispatched to registered PresenceListener objects.
     *
     * @param   emailAddress the email address to use to find presence info.
     */
    public void findPresence(String emailAddress)
    {
        // Make sure the service has been started.
        if (discovery != null)
        {
            // Send a remote discovery for presence information.
            discovery.getRemoteAdvertisements(null, DiscoveryService.ADV,
                tagEmailAddress, emailAddress, 0, null);

            // Do a local discovery for presence information.
            try
            {
                Enumeration enum = discovery.getLocalAdvertisements(
```

```
                DiscoveryService.ADV, tagEmailAddress, emailAddress);

            while (enum.hasMoreElements())
            {
                PresenceAdvertisement advertisement =
                    (PresenceAdvertisement) enum.nextElement();

                // Dispatch the advertisement to the registered presence
                // listeners.
                Enumeration listeners = registeredListeners.elements();
                while (listeners.hasMoreElements())
                {
                    PresenceListener listener =
                        (PresenceListener) listeners.nextElement();

                    // Notify the listener of the presence update.
                    listener.presenceUpdated(advertisement);
                }
            }
        }
        catch (IOException e)
        {
            System.out.println("Error in findPresence: " + e);
        }
    }
}

/**
 * Returns the advertisement for this service. In this case, this is
 * the ModuleImplAdvertisement passed in when the service was
 * initialized.
 *
 * @return  the advertisement describing this service.
 */
public Advertisement getImplAdvertisement()
{
    return implAdvertisement;
}

/**
 * Returns an interface used to protect this service.
 *
```

continues

Listing 11.6 **Continued**

```
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param        group the PeerGroup containing this service.
     * @param        assignedID the identifier for this service.
     * @param        implAdv the advertisement specifying this service.
     * @exception    PeerGroupException is not thrown ever by this
     *                implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;

        // Save a reference to the group of which that this service is
        // a part.
        peerGroup = group;

        // Get the local Peer ID.
        localPeerID = group.getPeerID().toString();

        // Register the advertisement type.
        // In some earlier versions of JXTA, registering the advertisement
        // doesn't work properly. To work around this, you can instead
        // simply instantiate the advertisement implementation directly.
        // This is not the recommended way to get an advertisement.
        // In this class, I've used the workaround in the discoveryEvent
        // and accouncePresence methods, but I've provided the
        // as well.
        /*
        AdvertisementFactory.registerAdvertisementInstance(
```

```
        PresenceAdvertisement.getAdvertisementType(),
            new PresenceAdv.Instantiator());
    */
}

/**
 * Remove a given listener object from the service. Once removed,
 * a listener will no longer be notified when a new Presence Response
 * Message arrives. This method is synchronized to prevent multiple
 * threads from altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to unregister.
 */
public synchronized boolean removeListener(PresenceListener listener)
{
    return registeredListeners.removeElement(listener);
}

/**
 * Start the service.
 *
 * @param   args the arguments to the service. Not used.
 * @return  0 to indicate the service started.
 */
public int startApp(String[] args)
{
    // Now that the service is being started, set the DiscoveryService
    // object to use to publish presence information.
    discovery = peerGroup.getDiscoveryService();

    // Add ourselves as a listener.
    discovery.addDiscoveryListener(this);

    return 0;
}

/**
 * Stop the service.
 */
public void stopApp()
{
    if (discovery != null)
```

continues

Listing 11.6 **Continued**

```
        {
            // Unregister ourselves as a listener.
            discovery.removeDiscoveryListener(this);
            discovery = null;

            // Empty the set of listeners.
            registeredListeners.removeAllElements();
        }
    }
}
```

Creating the Chat Service

Despite its name, the Chat service doesn't manage the chat session between two users. Instead, the Chat service is responsible for negotiating a Pipe Advertisement that can be used to establish a chat session. The Pipe Advertisement that is exchanged is used in conjunction with the `BidirectionalPipeService` to bind the input and output pipes to use from conducting the actual chat conversation.

The Initiate Chat Request Message

Before it can chat with a remote peer, a peer must request a Pipe Advertisement to establish the chat session with the remote peer. Although the Pipe Advertisement could have been included in the user's Presence Advertisement, there are a couple reasons for not doing this:

- **The functionality is unrelated**—Including the Pipe Advertisement in the Presence Advertisement would pollute the Presence Advertisement with information unrelated to conveying presence information. It would create an unnecessary link between the Presence service and the Chat service. Any developer who wanted to use the Chat service would end up having to incorporate the Presence service, even if the application didn't require Presence information.
- **A user wouldn't be able to restrict who can chat with him**—If a user's Presence Advertisement incorporated a Pipe Advertisement, anyone could start sending messages. By forcing a peer to request a Pipe Advertisement, the user's peer has the opportunity to block a chat session by not responding.

The Initiate Chat Request Message requests a Pipe Advertisement to use to establish a chat session using the XML shown in Listing 11.7.

Listing 11.7 **The Initiate Chat Request Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiateChatRequest>
  <EmailAddress> . . . </EmailAddress>
  <Name> . . . </Name>
</InitiateChatRequest>
```

The Initiate Chat Request Message contains the information that a peer receiving the request needs to determine whether to approve a chat session:

- **`EmailAddress`**—A required element containing the email address of the user making the request. The email address is used as a unique identifier for a user requesting a chat session.
- **`Name`**—An optional element containing a display name or common name for the user requesting the chat session.

When a peer receives an Initiate Chat Request Message, the peer can extract the `EmailAddress` and determine whether it wants to chat with the user requesting the chat session. If the peer wants to chat, an Initiate Chat Response Message is returned containing a Pipe Advertisement to use to establish the chat session. Otherwise, the peer does not return any response and the requesting peer cannot establish a chat session.

The Initiate Chat Request Message is broken into two classes, `InitiateChatRequestMessage` and `InitiateChatRequest`. The `InitiateChatRequestMessage` abstract class defines the majority of the functionality but leaves the implementation of the message rendering and parsing to the `InitiateChatRequest` class. The source code for `InitiateChatRequestMessage` and `InitiateChatRequest` is shown in Listings 11.8 and 11.9, respectively.

Listing 11.8 **Source Code for *InitiateChatRequestMessage.java***

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;


/**
 * An abstract class defining a request to begin a chat session. This
```

continues

Listing 11.8 **Continued**

```
 * request is responsible for sending information on a peer/user requesting
 * a chat session so that the recipient can use it to determine whether
 * to send a response containing a Pipe Advertisement to use to
 * start the chat session.
 */
public abstract class InitiateChatRequestMessage
{
    /**
     * The email address of the user requesting the chat session. This
     * is used to identify the user requesting the chat session.
     */
    private String emailAddress = null;

    /**
     * A display name to use to represent the user making the request
     * during the chat session.
     */
    private String name = null;


    /**
     * Returns a Document object containing the query's document tree.
     *
     * @param        asMimeType the desired MIME type for the query
     *               rendering.
     * @return       the Document containing the query's document object
     *               tree.
     */
    public abstract Document getDocument(MimeMediaType asMimeType);

    /**
     * Retrieve the email address of the user associated with the local
     * peer.
     *
     * @return  the email address used to identify the user.
     */
    public String getEmailAddress()
    {
        return emailAddress;
    }
```

```
    /**
     * Retrieve the display name of the user associated with the local peer.
     *
     * @return  the display name used to identify the user during the
     *          chat session.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Sets the email address of the user associated with the local peer.
     *
     * @param   emailAddress the email address used to identify the user.
     */
    public void setEmailAddress(String emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    /**
     * Sets the display name of the user associated with the local peer.
     *
     * @param   name the display name used to identify the user during
     *          the chat session.
     */
    public void setName(String name)
    {
        this.name = name;
    }
}
```

Listing 11.9 **Source Code for *InitiateChatRequest.java***

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;
```

continues

Listing 11.9 **Continued**

```
import java.util.Enumeration;

import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;


/**
 * An implementation of the InitiateChatRequestMessage abstract class. This
 * class is responsible for parsing and formatting the XML document used to
 * define a request to initiate a chat session.
 */
public class InitiateChatRequest extends InitiateChatRequestMessage
{
    /**
     * The root element for the request's XML document.
     */
    private static final String documentRootElement = "InitiateChatRequest";

    /**
     * A convenient constant for the XML MIME type.
     */
    private static final String mimeType = "text/xml";

    /**
     * The element name for the email address info.
     */
    private static final String tagEmailAddress = "EmailAddress";

    /**
     * The element name for the display name info.
     */
    private static final String tagName = "Name";
```

```
/**
 * Creates a new request object.
 */
public InitiateChatRequest()
{
    super();
}

/**
 * Creates a new Initiate Chat Request Message by parsing the
 * given stream.
 *
 * @param       stream the InputStream source of the query data.
 * @exception   IOException if the query can't be parsed from the
 *              stream.
 * @exception   IllegalArgumentException thrown if the data does not
 *              contain a Presence Query Message.
 */
public InitiateChatRequest(InputStream stream)
    throws IOException, IllegalArgumentException
{
    super();

    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);

    readDocument(document);
}

/**
 * Returns a Document object containing the request's document tree.
 *
 * @param       asMimeType the desired MIME type for the
 *              request rendering.
 * @return      the Document containing the request's document
 *              object tree.
 * @exception   IllegalArgumentException thrown if the email address
 *              is null.
 */
public Document getDocument(MimeMediaType asMimeType)
    throws IllegalArgumentException
```

continues

Listing 11.9 **Continued**

```
{
    // Check that the required elements are present.
    if (null != getEmailAddress())
    {
        StructuredDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                asMimeType, documentRootElement);
        Element element;

        element = document.createElement(tagEmailAddress,
            getEmailAddress());
        document.appendChild(element);

        element = document.createElement(tagName, getName());
        document.appendChild(element);

        return document;
    }
    else
    {
        throw new IllegalArgumentException("Missing email address");
    }
}

/**
 * Parses the given document tree for the request.
 *
 * @param       document the object containing the request data.
 * @exception   IllegalArgumentException if the document is not a chat
 *              request, as expected.
 */
public void readDocument(TextElement document)
    throws IllegalArgumentException
{
    if (document.getName().equals(documentRootElement))
    {
        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();
```

```
                if (element.getName().equals(tagEmailAddress))
                {
                    setEmailAddress(element.getTextValue());
                    continue;
                }

                if (element.getName().equals(tagName))
                {
                    setName(element.getTextValue());
                    continue;
                }
            }
        }
        else
        {
            throw new IllegalArgumentException(
                "Not a InitiateChatRequest document!");
        }
    }

    /**
     * Returns an XML String representation of the request.
     *
     * @return  the XML String representing this request.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc =
                (StructuredTextDocument) getDocument(
                    new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

The Initiate Chat Response Message

To allow a remote peer to establish a chat session, a peer that wants to grant a chat session must generate a Pipe Advertisement and send it as part of an Initiate Chat Response Message to the requesting peer. The XML for the Initiate Chat Response Message is shown in Listing 11.10.

Listing 11.10 **The Initiate Chat Response Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiateChatRequest>
  <EmailAddress> . . . </EmailAddress>
  <Name> . . . </Name>
  <jxta:PipeAdvertisement> . . . </jxta:PipeAdvertisement>
</InitiateChatRequest>
```

The Initiate Chat Response Message provides not only the Pipe Advertisement required to establish the chat session, but also other information that the recipient peer can use:

- **`EmailAddress`**—A required element containing the email address of the user approving the request for a chat session. The email address is used as a unique identifier for the user approving the chat session.
- **`Name`**—An optional element containing a display name or common name for the user approving the chat session.
- **`jxta:PipeAdvertisement`**—A required element that contains the Pipe Advertisement to use to establish the chat session. Note that this element is actually the root of the Pipe Advertisement XML tree.

When a peer receives an Initiate Chat Response Message, it can use the Pipe Advertisement with the `BidirectionalPipeService` class to establish two-way communications and begin chatting. By using the `BidirectionalPipeService`, you avoid having to create your own protocol to handle exchanging the Pipe Advertisements required to establish two-way communications. You need to create only one Pipe Advertisement, and the `BidirectionalPipeService` takes care of the details of exchanging Pipe Advertisements and binding input and output pipes.

The Initiate Chat Response Message is abstracted as two classes, `InitiateChatResponseMessage` and `InitiateChatResponse`. The `InitiateChatResponseMessage` abstract class defines the majority of the functionality but leaves the implementation of the message rendering and parsing to the `InitiateChatResponse` class. The source code for `InitiateChatResponseMessage` and `InitiateChatResponse` is shown in Listings 11.11 and 11.12, respectively.

Listing 11.11 **Source Code for *InitiateChatResponseMessage.java***

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;

import net.jxta.protocol.PipeAdvertisement;

/**
 * An abstract class defining a response to a request to begin a chat
 * session. This response is responsible for sending a Pipe Advertisement
 * in response to a Initiate Chat Request Message to allow a remote peer
 * to begin chatting with the local peer.
 */
public abstract class InitiateChatResponseMessage
{
    /**
     * The email address of the user associated with the local peer.
     * Used to map the user to presence information.
     */
    private String emailAddress = null;

    /**
     * A display name to use to represent the user associated with
     * the local peer during the chat session.
     */
    private String name = null;

    /**
     * A Pipe Advertisement to use to initiate the chat session. The
     * local peer will bind an input pipe to the pipe described by this
     * advertisement to set up the two-way chat communication channel
     * using the BidirectionalPipeService.
     */
    private PipeAdvertisement pipeAdvertisement = null;


    /**
     * Returns a Document object containing the response's document tree.
     *
     * @param       asMimeType the desired MIME type for the response
     *              rendering.
```

continues

Listing 11.11 **Continued**

```
     * @return      the Document containing the response's document
     *              object tree.
     */
    public abstract Document getDocument(MimeMediaType asMimeType);

    /**
     * Retrieve the email address of the user associated with the
     * local peer.
     *
     * @return  the email address used to identify the user.
     */
    public String getEmailAddress()
    {
        return emailAddress;
    }

    /**
     * Retrieve the display name of the user associated with the local peer.
     *
     * @return  the display name used to identify the user during the
     * chat session.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Returns the Pipe Advertisement object that a remote peer can use to
     * initiate the chat session.
     *
     * @return  the Pipe Advertisement to use for setting up the chat
     *          session.
     */
    public PipeAdvertisement getPipeAdvertisement()
    {
        return pipeAdvertisement;
    }

    /**
     * Sets the email address of the user associated with the local peer.
```

```
     *
     * @param   emailAddress the email address used to identify the user.
     */
    public void setEmailAddress(String emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    /**
     * Sets the display name of the user associated with the local peer.
     *
     * @param   name the display name used to identify the user during the
     *          chat session.
     */
    public void setName(String name)
    {
        this.name = name;
    }

    /**
     * Sets the Pipe Advertisement object that a remote peer can use to
     * initiate the chat session.
     *
     * @param   pipeAdvertisement the Pipe Advertisement to use for setting
     *          up the chat session.
     */
    public void setPipeAdvertisement(PipeAdvertisement pipeAdvertisement)
    {
        this.pipeAdvertisement = pipeAdvertisement;
    }
}
```

Listing 11.12 **Source Code for *InitiateChatResponse.java***

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;
```

continues

Listing 11.12 **Continued**

```
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;


/**
 * An implementation of the InitiateChatResponseMessage abstract class.
 * This class is responsible for parsing and formatting the XML document
 * used to define a response to a request to initiate a chat session.
 */
public class InitiateChatResponse extends InitiateChatResponseMessage
{
    /**
     * The root element for the response's XML document.
     */
    private static final String documentRootElement =
        "InitiateChatResponse";

    /**
     * A convenient constant for the XML MIME type.
     */
    private static final String mimeType = "text/xml";

    /**
     * The element name for the display name info.
     */
    private static final String tagName = "Name";

    /**
     * The element name for the email address info.
     */
```

```
private static final String tagEmailAddress = "EmailAddress";


/**
 * Creates new response object.
 */
public InitiateChatResponse()
{
    super();
}

/**
 * Creates a new Initiate Chat Response Message by parsing the given
 * stream.
 *
 * @param       stream the InputStream source of the response data.
 * @exception   IOException if the response can't be parsed from the
 *              stream.
 * @exception   IllegalArgumentException thrown if the data does not
 *              contain a Presence Response Message.
 */
public InitiateChatResponse(InputStream stream) throws IOException,
    IllegalArgumentException
{
    super();

    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);

    readDocument(document);
}

/**
 * Returns a Document object containing the response's document tree.
 *
 * @param       asMimeType the desired MIME type for the response
 *              rendering.
 * @return      the Document containing the response's document
 *              object tree.
 * @exception   IllegalArgumentException thrown if the Pipe
 *              Advertisement
```

continues

Listing 11.12 **Continued**

```
     *               or the name is null.
     */
    public Document getDocument(MimeMediaType asMimeType)
        throws IllegalArgumentException
    {
        // Check that the required elements are present.
        if ((null != getPipeAdvertisement()) && (null != getName()))
        {
            StructuredDocument document = (StructuredTextDocument)
                StructuredDocumentFactory.newStructuredDocument(
                    asMimeType, documentRootElement);
            Element element;

            PipeAdvertisement pipeAdv = getPipeAdvertisement();
            if (pipeAdv != null)
            {
                StructuredTextDocument advDoc = (StructuredTextDocument)
                    pipeAdv.getDocument(asMimeType);
                StructuredDocumentUtils.copyElements(
                    document, document, advDoc);
            }

            element = document.createElement(tagName, getName());
            document.appendChild(element);

            element = document.createElement(tagEmailAddress,
                getEmailAddress());
            document.appendChild(element);

            return document;
        }
        else
        {
            throw new IllegalArgumentException("Missing pipe ID or name!");
        }
    }

    /**
     * Parses the given document tree for the response.
     *
     * @param       document the object containing the response data.
```

```
 * @exception   IllegalArgumentException if the document is not a
 *              response, as expected.
 */
public void readDocument(TextElement document)
    throws IllegalArgumentException
{
    if (document.getName().equals(documentRootElement))
    {
        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();

            if (element.getName().equals(tagName))
            {
                setName(element.getTextValue());
                continue;
            }

            if (element.getName().equals(tagEmailAddress))
            {
                setEmailAddress(element.getTextValue());
                continue;
            }

            if (element.getName().equals(
                PipeAdvertisement.getAdvertisementType()))
            {
                try
                {
                    PipeAdvertisement pipeAdv = (PipeAdvertisement)
                        AdvertisementFactory.newAdvertisement(element);
                    setPipeAdvertisement( pipeAdv );
                }
                catch ( ClassCastException wrongAdv )
                {
                    throw new IllegalArgumentException(
                        "Bad pipe advertisement in advertisement" );
                }

                continue;
```

continues

Listing 11.12 **Continued**

```
                }
            }
        }
        else
        {
            throw new IllegalArgumentException(
                "Not a InitiateChatResponse document!");
        }
    }

    /**
     * Returns an XML String representation of the response.
     *
     * @return  the XML String representing this response.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
                getDocument(new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

The Chat Message

Although the Chat service doesn't handle sending and receiving chat messages, this is probably the most appropriate place to mention the message used to send a chat message to a remote user. You could create a class to encapsulate the chat message, but in this simple implementation, only one piece of information needs to be sent to a remote user: the chat message text itself.

To send a chat message to a remote user after the pipes have been established, a peer only needs to create a new `Message` object and populate a message element named `ChatMessage` with the chat message text.

The Chat Service

The Chat service abstracts the creation of Initiate Chat Request and Response Messages and provides a simple interface that a developer can use to send these messages. As with `PresenceService`, the `ChatService` interface shown in Listing 11.13 provides a mechanism for developers to register and unregister listener objects that can be used to handle the requests and responses.

Listing 11.13 **Source Code for *ChatService.java***

```
package com.newriders.jxta.chapter11.chat;

import net.jxta.protocol.PipeAdvertisement;

import net.jxta.service.Service;


/**
 * An interface for the Chat service, a service that allows peers to
 * request and approve chat sessions. This interface defines the operations
 * that a developer can expect to use to manipulate the Chat service,
 * regardless of which underlying implementation of the service is being
 * used.
 */
public interface ChatService extends Service
{
    /**
     * The module class ID for the Presence class of service.
     */
    public static final String refModuleClassID =
        "urn:jxta:uuid-F84F9397891240B496D1B5754CCC933105";


    /**
     * Add a listener object to the service. When a new Initiate Chat
     * Request or Response Message arrives, the service will notify each
     * registered listener.
     *
     * @param   listener the listener object to register with the service.
     */
```

continues

Listing 11.13 **Continued**

```
    public void addListener(ChatListener listener);

    /**
     * Approve a chat session.
     *
     * @param   pipeAdvertisement the advertisement for the pipe that will
     *          be used to set up the chat session.
     * @param   emailAddress the emailAddress of the user associated with
     *          local peer.
     * @param   displayName the name of the user associated with the
     *          local peer.
     * @param   queryID the query ID to use to send to the Resolver
     *          Response Message containing the response, allowing the
     *          remote peer to match the response to an initial request.
     */
    public void approveChat(PipeAdvertisement pipeAdvertisement,
        String emailAddress, String displayName, int queryID);

    /**
     * Removes a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new Initiate Chat
     * Request or Response Message arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public boolean removeListener(ChatListener listener);

    /**
     * Send a request to chat to the peer specified.
     *
     * @param   peerID the Peer ID of the remote peer to request for a chat
     *          session.
     * @param   emailAddress the email address of the user associated with
     *          the local peer.
     * @param   displayName the display name  of the user associated with
     *          the local peer.
     * @param   listener the listener to notify when a response to this
     *          request is received.
     */
    public void requestChat(String peerID, String emailAddress,
        String displayName, ChatListener listener);
}
```

For this application, we would like the main application to be capable of determining whether a request to chat should be approved. This allows the application to ignore a request from a user that isn't a part of the user's list of contacts. To accomplish this, the `ChatService` delegates the decision of whether to approve a chat request using the `ChatListener` interface shown in Listing 11.14.

Listing 11.14 **Source Code for *ChatListener.java***

```
package com.newriders.jxta.chapter11.chat;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;
import com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;


/**
 * An interface to encapsulate an object that listens for notification
 * from the ChatService of newly arrived requests for a chat session and
 * responds to requests for a chat session.
 */
public interface ChatListener
{
    /**
     * Notify the listener that a chat session has been approved.
     *
     * @param   response the response to the request for a chat session.
     */
    public void chatApproved(InitiateChatResponseMessage response);

    /**
     * Notify the listener that a chat session has been requested.
     *
     * @param   request the object containing the chat session request info.
     * @param   queryID the query ID from the Resolver Query Message used to
     *          send the request.
     */
    public void chatRequested(InitiateChatRequestMessage request,
        int queryID);
}
```

When the `ChatService` receives an Initiate Chat Request Message, it notifies each of the registered `ChatListener` instance's `chatRequested` methods. It is the responsibility of a listener to approve a request. When the `ChatService` receives

an Initiate Chat Response Message, the registered `ChatListener` instance's `chatApproved` method handles the response, and uses its contents to begin the chat session.

For this application, the `ChatService` implementation shown in Listing 11.15 uses the Resolver service to handle the Initiate Chat Request and Response Messages.

Listing 11.15 **Source Code for *ChatServiceImpl.java***

```
package com.newriders.jxta.chapter11.impl.chat;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Hashtable;
import java.util.Vector;

import net.jxta.document.Advertisement;

import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.NoResponseException;
import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ResendQueryException;

import net.jxta.id.ID;

import net.jxta.impl.protocol.ResolverQuery;
import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.peergroup.PeerGroup;

import net.jxta.pipe.PipeID;

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;

import net.jxta.service.Service;
```

```
import com.newriders.jxta.chapter11.chat.ChatListener;
import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.protocol.InitiateChatRequest;
import com.newriders.jxta.chapter11.impl.protocol.InitiateChatResponse;


/**
 * The implementation of the ChatService interface. This service
 * builds on top of the Resolver service to provide the functionality
 * for requesting and approving a chat session.
 */
public class ChatServiceImpl implements ChatService, QueryHandler
{
    /**
     * The Module Specification ID for the Chat service.
     */
    public static final String refModuleSpecID =
        "urn:jxta:uuid-F84F9397891240B496D1B5754CCC9331DFFD"
        + "10CDD5A140A6B8A1BC18CD65582106";

    /**
     * The set of listener objects registered with the service
     * to handle an approval to start a chat session. These
     * listeners are associated with a specific query ID used
     * to send a request to start a chat session to a remote user.
     */
    private Hashtable approvedListeners = new Hashtable();

    /**
     * The handler name used to register the Resolver handler.
     */
    private String handlerName = null;

    /**
     * The Module Implementation advertisement for this service.
     */
    private Advertisement implAdvertisement = null;

    /**
     * The local Peer ID.
     */
    private String localPeerID = null;
```

continues

Listing 11.15 **Continued**

```
/**
 * The peer group to which the service belongs.
 */
private PeerGroup peerGroup = null;

/**
 * A unique query ID that can be used to track a query.
 * This is constant across instances of the service on the
 * same peer to ensure queryID uniqueness for the peer.
 */
private static int queryID = 0;

/**
 * The set of listener objects registered with the service
 * to handle requests to start a chat session.
 */
private Vector requestListeners = new Vector();

/**
 * The Resolver service used to handle queries and responses.
 */
private ResolverService resolver = null;


/**
 * Create a new ChatServiceImpl object.
 */
public ChatServiceImpl()
{
    super();
}

/**
 * Add a listener object to the service. When a new Initiate Chat
 * Request or Response Message arrives, the service will notify each
 * registered listener. This method is synchronized to prevent multiple
 * threads from altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
```

```
public synchronized void addListener(ChatListener listener)
{
    requestListeners.addElement(listener);
}

/**
 * Approve a chat session.
 *
 * @param   pipeAdvertisement the advertisement for the pipe that will
 *          be used to set up the chat session.
 * @param   emailAddress the emailAddress of the user associated with
 *          local peer.
 * @param   displayName the name of the user associated with the local
 *          peer.
 * @param   queryID the query ID to use to send to the Resolver Response
 *          Message containing the response, allowing the remote peer to
 *          match the response to an initial request.
 */
public void approveChat(PipeAdvertisement pipeAdvertisement,
    String emailAddress, String displayName, int queryID)
{
    // Make sure that the service has been started.
    if (resolver != null)
    {
        ResolverResponse response;

        // Create the response message and populate it with the
        // given Pipe ID.
        InitiateChatResponse reply = new InitiateChatResponse();
        reply.setPipeAdvertisement(pipeAdvertisement);
        reply.setEmailAddress(emailAddress);
        reply.setName(displayName);

        // Wrap the response message in a resolver response message.
        response = new ResolverResponse(handlerName, "JXTACRED",
            queryID, reply.toString());

        // Send the request using the Resolver service.
        resolver.sendResponse(null, response);
    }
}

/**
```

continues

Listing 11.15 **Continued**

```
     * Returns the advertisement for this service. In this case, this is
     * the ModuleImplAdvertisement passed in when the service was
     * initialized.
     *
     * @return  the advertisement describing this service.
     */
    public Advertisement getImplAdvertisement()
    {
        return implAdvertisement;
    }

    /**
     * Returns an interface used to protect this service.
     *
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param       group the PeerGroup containing this service.
     * @param       assignedID the identifier for this service.
     * @param       implAdv the advertisement specifying this service.
     * @exception   PeerGroupException is not thrown ever by this
     *              implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save a reference to the group of which that this service is
        // a part.
        peerGroup = group;

        // Use the assigned ID as the Resolver handler name.
        handlerName = assignedID.toString();
```

```
        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;

        // Get the local Peer ID.
        localPeerID = group.getPeerID().toString();
    }

    /**
     * Process a Resolver Query Message.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
        throws IOException, NoResponseException, DiscardQueryException,
            ResendQueryException
    {
        ResolverResponse response;
        InitiateChatRequest request;

        try
        {
            // Extract the request message.
            request = new InitiateChatRequest(
                new ByteArrayInputStream((query.getQuery()).getBytes()));
        }
        catch (Exception e)
        {
            // Not the expected format of the message.
            throw new NoResponseException();
        }

        // Notify each of the registered listeners.
        if (requestListeners.size() > 0)
        {
            ChatListener listener = null;

            for (int i = 0; i < requestListeners.size(); i++)
            {
                listener = (ChatListener) requestListeners.elementAt(i);
                listener.chatRequested(request, query.getQueryId());
            }
        }

        // Throw NoResponseException because this service will not
        // produce a InitiateChatResponse. It's the responsibility of a
```

continues

Listing 11.15 **Continued**

```
        // ChatListener to decide whether to accept the request to chat
        // and inform the requestor of the Pipe ID to use to send chat
        // messages.
        throw new NoResponseException();
    }

    /**
     * Process a Resolver response message.
     *
     * @param   response a response message to be processed.
     */
    public void processResponse(ResolverResponseMsg response)
    {
        InitiateChatResponse reply;
        ChatListener listener = null;
        String responseString = response.getResponse();

        if (null != responseString)
        {
            try
            {
                // Extract the message from the Resolver response.
                reply = new InitiateChatResponse(
                    new ByteArrayInputStream(responseString.getBytes()));

                // Notify the listener associated with the response's
                // queryID.
                listener = (ChatListener) approvedListeners.get(
                    new Integer(response.getQueryId()));
                if (listener != null)
                {
                    listener.chatApproved(reply);
                }
            }
            catch (Exception e)
            {
                // This is not the right type of response message, or
                // the message is improperly formed. Ignore the exception;
                // do nothing with the message.
                System.out.println("Error in response: " + e);
            }
```

```
    }
}

/**
 * Remove a given listener object from the service. Once removed,
 * a listener will no longer be notified when a new Initiate Chat
 * Request or Response Message arrives. This method is synchronized to
 * prevent multiple threads from altering the set of registered
 * listeners simultaneously.
 *
 * @param   listener the listener object to unregister.
 */
public synchronized boolean removeListener(ChatListener listener)
{
    return requestListeners.removeElement(listener);
}

/**
 * Send a request to chat to the peer specified.
 *
 * @param   peerID the Peer ID of the remote peer to request for a
 *          chat session.
 * @param   emailAddress the email address of the user associated
 *          with the local peer.
 * @param   displayName the display name of the user associated with
 *          the local peer.
 * @param   listener the listener to notify when a response to this
 *          request is received.
 */
public void requestChat(String peerID, String emailAddress,
    String displayName, ChatListener listener)
{
    // Make sure that the service has been started.
    if (resolver != null)
    {
        // Create the request object.
        String localPeerID = peerGroup.getPeerID().toString();
        InitiateChatRequest request = new InitiateChatRequest();

        // Configure the request.
        request.setEmailAddress(emailAddress);
        request.setName(displayName);
```

continues

Listing 11.15 **Continued**

```
            // Wrap the query in a Resolver Query Message.
            ResolverQuery query = new ResolverQuery(handlerName,
                "JXTACRED", localPeerID, request.toString(), queryID++);

            // Add the given listener to the set of approved listeners.
            // This will be used to ensure that only responses to actual
            // queries sent by this service will be passed to the given
            // listener.
            approvedListeners.put(
                new Integer(query.getQueryId()), listener);

            // Send the request to the peer using the Resolver service.
            resolver.sendQuery(peerID, query);
        }
    }

    /**
     * Start the service.
     *
     * @param   args the arguments to the service. Not used.
     * @return  0 to indicate the service started.
     */
    public int startApp(String[] args)
    {
        // Now that the service is being started, set the ResolverService
        // object to use handle our queries and send responses.
        resolver = peerGroup.getResolverService();

        // Add ourselves as a handler using the uniquely constructed
        // handler name.
        resolver.registerHandler(handlerName, this);

        return 0;
    }

    /**
     * Stop the service.
     */
```

```
    public void stopApp()
    {
        if (resolver != null)
        {
            // Unregister ourselves as a listener.
            resolver.unregisterHandler(handlerName);
            resolver = null;

            // Empty the set of request and approved listeners.
            requestListeners.removeAllElements();
        }
    }
}
```

The JXTA Messenger Application

The JXTA Messenger application is the chat application that the end user will see and use to conduct a chat session with a remote user. The application itself consists of two pieces: an application module to show the user interface and a main application to handle configuring and creating the peer group.

The User Interface

The `ExampleService` example developed in Chapter 10, "Peer Groups and Services," had no user interface of its own to allow a user to interact with the service. The `ExampleServiceTest` class provided the user interface and interacted with the peer group's `ExampleService` instance to provide functionality. Although you could do the same thing in the JXTA Messenger, it would be better to wrap up the entire user interface as an application that starts when the peer group starts.

Fortunately, the reference implementation of JXTA provides the capability to add an application to a peer group's Module Implementation Advertisement parameters. When the `ExampleService` implementation was added to the service parameters for the peer group created in `ExampleServiceTest`, it involved code similar to the code shown in Listing 11.16.

Listing 11.16 **Adding a Service to the Peer Group Parameters**

```
ModuleImplAdvertisement implAdv =
    netPeerGroup.getAllPurposePeerGroupImplAdvertisement();
StdPeerGroupParamAdv params = new StdPeerGroupParamAdv(implAdv.getParam());
Hashtable services = params.getServices();
services.put(moduleClassAdv.getModuleClassID(), moduleImplAdv);
params.setServices(services);
implAdv.setParam((StructuredDocument) params.getDocument(
    new MimeMediaType("text", "xml")));
```

This code added the service specified by the `moduleImplAdv` Module Implementation Advertisement to the set of parameters in `implAdv`, the peer group's Module Implementation Advertisement. The new service is added to the parameters' `services Hashtable` using the service's Module Class ID as a key.

In the reference implementation, any class that implements the `net.jxta.platform.Application` interface can be added to the peer group's Module Implementation Advertisement parameters in a similar fashion. Instead of adding to the parameters' set of services, add the application to the parameters' set of applications.

```
Hashtable applications = params.getApps ();
. . .
params.setApps(applications);
```

Only two real differences exist between an application module and a service module in the reference implementation:

- **Which interface the module implements**—An application module implements the `net.jxta.platform.Application` interface, whereas a service module implements the `net.jxta.service.Service` interface. Both interfaces extend the `net.jxta.platform.Module` interface, and only `Service` adds methods not found in `Module`.
- **When the module's `startApp` method is called**—Each of a peer group's services has its `startApp` method called when the peer group, which is also a module, is initialized using the `PeerGroup.init` method. Each of the peer group's applications has its `startApp` method called when the peer group is started using the `PeerGroup.startApp` method.

So, rather than try to have the main application figure out when the peer group has started, you should place the user interface in a separate class that implements the `Application` interface. That way, the user interface automatically is notified via its `startApp` method when the peer group containing the Presence and Chat services has been started.

The `BuddyList` class, shown in Listing 11.17, handles the main user interface for the JXTA Messenger. The `BuddyList` provides a user interface that displays a list of buddies that the user is monitoring for presence information.

Listing 11.17 **Source Code for *BuddyList.java***

```
package com.newriders.jxta.chapter11;

import java.util.Enumeration;
import java.io.IOException;

import java.util.Hashtable;

import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.UnknownServiceException;

import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.DefaultListModel;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;

import javax.swing.border.Border;

import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import net.jxta.document.Advertisement;
```

continues

Listing 11.17 **Continued**

```
import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ServiceNotFoundException;

import net.jxta.id.ID;
import net.jxta.id.IDFactory;

import net.jxta.impl.util.BidirectionalPipeService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.platform.Application;
import net.jxta.platform.ModuleClassID;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.chat.ChatListener;
import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.protocol.InitiateChatResponse;

import com.newriders.jxta.chapter11.presence.PresenceListener;
import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;
import com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;
import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * A user interface application to show a buddy list and their current
 * status using the Presence service. The application also allows the user
 * to initiate a chat session using the Chat service.
 */
public class BuddyList extends JFrame implements Application,
    ActionListener, PresenceListener
{
    /**
     * The Module Class ID for the BuddyList application.
     */
    public static final String refModuleClassID =
        "urn:jxta:uuid-E340D55F97E141C9B46FB2B108D8C2B705";
```

```
/**
 * The Module Specification ID for the BuddyList application.
 */
public static final String refModuleSpecID =
    "urn:jxta:uuid-E340D55F97E141C9B46FB2B108D8C2B7"
    + "581B0312E66046B6BB96BF6A2EC5F27906";

/**
 * A list to display the set of "approved" buddies and their
 * presence status.
 */
private JList buddies = new JList();

/**
 * The data model for the list widget.
 */
private DefaultListModel buddiesData = new DefaultListModel();

/**
 * The peer group to which the service belongs.
 */
private PeerGroup peerGroup = null;

/**
 * The Presence service used to update other users of this user's
 * current presence status.
 */
private PresenceService presence = null;

/**
 * The Chat service used to handle requesting chat sessions and
 * respond with approvals.
 */
private ChatService chat = null;

/**
 * A set of buddy email addresses, indexed by display name.
 */
private Hashtable buddyNames = new Hashtable();

/**
 * A set of buddy display names, indexed by email address.
 */
```

continues

Listing 11.17 **Continued**

```
private Hashtable buddyEmailAddresses = new Hashtable();

/**
 * A set of buddy Peer IDs, indexed by email address.
 */
private Hashtable buddyPeerIDs = new Hashtable();

/**
 * The local user's email address.
 */
private String emailAddress = null;

/**
 * The local user's display name.
 */
private String displayName = null;

/**
 * The local user's current presence status.
 */
private int presenceStatus = PresenceService.OFFLINE;

/**
 * A handler to use for handling chat requests and approvals.
 */
private ChatHandler chatHandler = new ChatHandler();


/**
 * A simple menu handler to deal with the "Set Name..." menu item.
 */
public class SetNameHandler implements ActionListener
{
    /**
     * Handles the "Set Name..." menu item.
     *
     * @param   e the event for the menu item.
     */
    public void actionPerformed(ActionEvent e)
    {
        String newDisplayName = JOptionPane.showInputDialog(null,
```

```
                "Enter a new display name :",
                "Set Display Name...", JOptionPane.QUESTION_MESSAGE);

            if ((null != newDisplayName) && (0 < newDisplayName.length()))
            {
                // Announce change in presence status.
                presence.announcePresence(
                    presenceStatus, emailAddress, newDisplayName);

                displayName = newDisplayName;
            }
        }
    }

    /**
     * A simple menu handler to deal with the "Add Buddy..." menu item.
     */
    public class AddBuddyHandler implements ActionListener
    {
        /**
         * Handles the "Add Buddy..." menu item.
         *
         * @param   e the event for the menu item.
         */
        public void actionPerformed(ActionEvent e)
        {
            String buddy = JOptionPane.showInputDialog(null,
                "Enter the email address of your buddy:",
                "Add Buddy...", JOptionPane.QUESTION_MESSAGE);

            if ((null != buddy) && (0 < buddy.length()))
            {
                // Ensure that the buddy isn't already in our list.
                if (null == buddyEmailAddresses.get(buddy))
                {
                    // We should really validate the email address, but
                    // for simplicity we'll just add it to the list
                    // marking the buddy as "offline". Use the email
                    // address as the buddy display name until we discover
                    // presence information.
                    add(buddy, buddy, PresenceService.OFFLINE);

                    // Find the presence information for the buddy.
```

continues

Listing 11.17 **Continued**

```
                presence.findPresence(buddy);
            }
            else
            {
                JOptionPane.showMessageDialog(null,
                    "A buddy with that email address already exists!",
                    "Buddy Exists!", JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}

/**
 * A handler to deal with the application dialog being closed or the
 * "Quit" menu item being activated.
 */
public class QuitHandler extends WindowAdapter implements ActionListener
{
    /**
     * Handles the "Quit" menu item.
     *
     * @param   e the event for the menu item.
     */
    public void actionPerformed(ActionEvent e)
    {
        quit();
    }

    /**
     * Handles the window's system Close button.
     *
     * @param   e the event for the window closing.
     */
    public void windowClosing(WindowEvent e)
    {
        quit();
    }

    /**
     * Quits the application.
     */
```

```
    private void quit()
    {
        // Announce change in presence status.
        presenceStatus = PresenceService.OFFLINE;
        presence.announcePresence(
            presenceStatus, emailAddress, displayName);

        System.exit(0);
    }
}

/**
 * Handles the local user updating presence information.
 */
public class StatusChangeHandler implements ActionListener
{
    /**
     * Handles the "My Status" submenu items.
     *
     * @param   e the event for the menu item.
     */
    public void actionPerformed(ActionEvent e)
    {
        int newPresenceStatus = PresenceService.OFFLINE;
        String presenceString =
            ((JCheckBoxMenuItem) e.getSource()).getText();

        if (presenceString.equals("Offline"))
        {
            newPresenceStatus = PresenceService.OFFLINE;
        }
        else if (presenceString.equals("Online"))
        {
            newPresenceStatus = PresenceService.ONLINE;
        }
        else if (presenceString.equals("Busy"))
        {
            newPresenceStatus = PresenceService.BUSY;
        }
        else if (presenceString.equals("Away"))
        {
            newPresenceStatus = PresenceService.AWAY;
        }
```

continues

Listing 11.17 **Continued**

```
            if (newPresenceStatus != presenceStatus)
            {
                // Announce change in presence status.
                presence.announcePresence(
                    newPresenceStatus, emailAddress, displayName);
                presenceStatus = newPresenceStatus;
            }
        }
    }

    /**
     * A simple handler to deal with spawning a chat window when a chat
     * request is approved or for handling incoming chat requests.
     */
    public class ChatHandler implements ChatListener
    {
        /**
         * Handles an approval for a previously generated chat request.
         * Displays the ChatDialog and handles establishing the two-
         * way communication channel.
         *
         * @param   response the response object containing the Pipe
         *          Advertisement to use to establish two-way communication.
         */
        public void chatApproved(InitiateChatResponseMessage response)
        {
            ChatDialog chatDialog = null;

            // Extract the Pipe Advertisement from the chat response.
            PipeAdvertisement pipeAdv = response.getPipeAdvertisement();

            if (null != pipeAdv)
            {
                // Create a bidirectional pipe.
                BidirectionalPipeService pipeService =
                    new BidirectionalPipeService(peerGroup);
                BidirectionalPipeService.Pipe pipe = null;
                String buddyName = null;

                while (null == pipe)
```

```
        {
            // We just loop here.
            try
            {
                pipe = pipeService.connect(pipeAdv, 120000);
            }
            catch (IOException e)
            {
                // Do nothing.
                System.out.println("Connect error:" + e);
            }
        }

        // Get the buddy's display name.
        buddyName = response.getName();
        if (buddyName == null)
        {
            buddyName = response.getEmailAddress();
        }

        // Create the conversation GUI, and show it.
        chatDialog = new ChatDialog(buddyName, displayName,
            peerGroup.getPipeService(), pipe.getInputPipe(),
            pipe.getOutputPipe());
        chatDialog.show();
    }
    else
    {
        JOptionPane.showMessageDialog(null,
            "Buddy's reply is missing pipe advertisement!",
            "Unable To Chat!", JOptionPane.ERROR_MESSAGE);
    }
}

/**
 * Handles an incoming request for a chat session. Checks that
 * the incoming request comes from a known buddy and responds
 * with an approval message. Also prepares two-way communications
 * and the chat user interface.
 *
 * @param   request the request for a chat session.
 * @param   queryID the query ID to be used to send a response
 *          using the Resolver service.
```

continues

Listing 11.17 **Continued**

```
 */
public void chatRequested(InitiateChatRequestMessage request,
    int queryID)
{
    String buddyEmailAddress = request.getEmailAddress();

    // Check who is making the request against the list of
    // approved chat buddies.
    if (null != buddyEmailAddresses.get(buddyEmailAddress))
    {
        ChatDialog chatDialog = null;
        String buddyName = null;

        // If the request is part of the approved buddies, then
        // approve the chat request.
        BidirectionalPipeService pipeService =
            new BidirectionalPipeService(peerGroup);
        BidirectionalPipeService.Pipe pipe = null;

        // Create an accept pipe to use to create an input pipe and
        // listen for connections.
        try
        {
            BidirectionalPipeService.AcceptPipe acceptPipe =
                pipeService.bind("JXTA Messenger Pipe");

            // Extract the Pipe Advertisement and the Pipe ID.
            PipeAdvertisement pipeAdv =
                acceptPipe.getAdvertisement();

            // Send the approval response.
            chat.approveChat(pipeAdv, emailAddress,
                displayName, queryID);

            // "Accept" a connection, meaning set up the input pipe
            // and listen for messages. Set this object as the
            // MessageListener so that we can handle incoming
            // messages without having to spawn a thread to call
            // waitForMessage on the input pipe.
            while (null == pipe)
            {
                // We just loop here.
```

```
                    try
                    {
                        pipe = acceptPipe.accept(1200000);
                    }
                    catch (InterruptedException e)
                    {
                        // Do nothing.
                        System.out.println("Interrupted: " + e);
                    }
                }

                // Get the buddy's display name.
                buddyName = request.getName();
                if (buddyName == null)
                {
                    buddyName = request.getEmailAddress();
                }

                // Create the conversation GUI.
                chatDialog = new ChatDialog(buddyName, displayName,
                    peerGroup.getPipeService(), pipe.getInputPipe(),
                    pipe.getOutputPipe());
            }
            catch (IOException e2)
            {
                   System.out.println("Error in chatRequested: " + e2);
            }
        }
    }
}

/**
 * Creates a new BuddyList object.
 */
public BuddyList()
{
    super("JXTA Messenger");
}

/**
 * Handles the user interface event used to trigger
 * a chat session with a buddy from the list.
```

continues

Listing 11.17 **Continued**

```
     *
     * @param   e the event being handled.
     */
    public void actionPerformed(ActionEvent e)
    {
        if (null != chat)
        {
            if (-1 != buddies.getSelectedIndex())
            {
                String buddyEmailAddress = null;
                String peerID = null;
                String buddyName =
                    (String) buddiesData.get(buddies.getSelectedIndex());

                // Extract the actual buddy name (without the
                // presence string).
                buddyName = buddyName.substring(0, buddyName.indexOf(" ("));

                // Figure out the email address and Peer ID.
                buddyEmailAddress = (String) buddyNames.get(buddyName);
                peerID = (String) buddyPeerIDs.get(buddyEmailAddress);

                if ((null != buddyEmailAddress) && (null != peerID))
                {
                    // It would be best to time out due to lack of approval
                    // for chat session, but for simplicity we'll just
                    // hope that the request gets through. In a full-fledged
                    // application, it would be better to time out and retry.
                    chat.requestChat(peerID, emailAddress, displayName,
                        chatHandler);
                }
                else
                {
                    JOptionPane.showMessageDialog(null,
                        "Could not find buddy!", "Unable To Find Buddy",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
            else
            {
                JOptionPane.showMessageDialog(null,
```

```
                "No buddy selected!", "Select A Buddy First!",
                JOptionPane.ERROR_MESSAGE);
        }
    }
}

/**
 * Add a buddy to the list of buddies.
 *
 * @param   buddyDisplayName the display name for the buddy.
 * @param   buddyEmailAddress the email address for the buddy.
 * @param   status the initial presence status for the buddy.
 */
public void add(String buddyDisplayName, String buddyEmailAddress,
    int status)
{
    String buddyStatus;

    buddyNames.put(buddyDisplayName, buddyEmailAddress);
    buddyEmailAddresses.put(buddyEmailAddress, buddyDisplayName);

    // Construct the list entry from both the displayName and
    // the current status.
    buddyStatus = buddyDisplayName + " ";
    switch (status)
    {
        case PresenceService.OFFLINE:
        {
            buddyStatus += "(Offline)";
            break;
        }

        case PresenceService.ONLINE:
        {
            buddyStatus += "(Online)";
            break;
        }

        case PresenceService.BUSY:
        {
            buddyStatus += "(Busy)";
            break;
        }
```

continues

Listing 11.17 **Continued**

```
            case PresenceService.AWAY:
            {
                buddyStatus += "(Away)";
                break;
            }

            default:
            {
                buddyStatus += "(Offline)";
                break;
            }
        }
        buddiesData.addElement(buddyStatus);
    }

    /**
     * Initialize the BuddyList application for the peer group.
     *
     * @param        group the peer group containing this application.
     * @param        ID the assigned ID for the application.
     * @param        implAdv the Module Implementation Advertisement for
     *               the BuddyList application.
     * @exception    PeerGroupException never thrown in this implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        this.peerGroup = group;

        // Initialize the user interface for the buddy list.
        initializeUserInterface();
    }

    /**
     * Initialize the menu for the BuddyList user interface.
     */
    public JMenuBar initializeMenu()
    {
        JMenuBar menuBar = new JMenuBar();
        JMenu actionsMenu = new JMenu("Actions");
        JMenu myStatusMenuItem = new JMenu("My Status");
```

```
        JCheckBoxMenuItem offline = new JCheckBoxMenuItem("Offline");
        JCheckBoxMenuItem online = new JCheckBoxMenuItem("Online", true);
        JCheckBoxMenuItem busy = new JCheckBoxMenuItem("Busy");
        JCheckBoxMenuItem away = new JCheckBoxMenuItem("Away");
        JMenuItem addBuddyMenuItem = new JMenuItem("Add Buddy...");
        JMenuItem chatBuddyMenuItem = new JMenuItem("Chat With Buddy...");
        JMenuItem setNameMenuItem = new JMenuItem("Set Name...");
        JMenuItem quitMenuItem = new JMenuItem("Quit");
        ButtonGroup group = new ButtonGroup();

        // Configure the status menu.
        myStatusMenuItem.add(offline);
        group.add(offline);
        myStatusMenuItem.add(online);
        group.add(online);
        myStatusMenuItem.add(busy);
        group.add(busy);
        myStatusMenuItem.add(away);
        group.add(away);

        // Configure the listeners for the menu items.
        addBuddyMenuItem.addActionListener(new AddBuddyHandler());
        chatBuddyMenuItem.addActionListener(this);
        setNameMenuItem.addActionListener(new SetNameHandler());
        quitMenuItem.addActionListener(new QuitHandler());
        offline.addActionListener(new StatusChangeHandler());
        online.addActionListener(new StatusChangeHandler());
        busy.addActionListener(new StatusChangeHandler());
        away.addActionListener(new StatusChangeHandler());

        // Add the menu items to the menus.
        actionsMenu.add(myStatusMenuItem);
        actionsMenu.add(addBuddyMenuItem);
        actionsMenu.add(chatBuddyMenuItem);
        actionsMenu.add(setNameMenuItem);
        actionsMenu.addSeparator();
        actionsMenu.add(quitMenuItem);

        // Add the menus to the menu bar.
        menuBar.add(actionsMenu);

        return menuBar;
    }
```

continues

Listing 11.17 **Continued**

```
/**
 * Initialize the main user interface for the BuddyList application.
 */
public void initializeUserInterface()
{
    Container framePanel = getContentPane();

    // Set a border for the list of buddies.
    Border border = BorderFactory.createTitledBorder("Buddies");

    // Configure menu.
    setJMenuBar(initializeMenu());

    // Add the initialized inner panel to the frame panel.
    framePanel.add(buddies);

    // Pack the frame, preparing it for display.
    pack();
    setSize(200, 300);

    // Add a listener to handle quitting the application when
    // the window is closed using the system menu.
    addWindowListener(new QuitHandler());

    // Add ourselves as a listener to the list, and set
    // the model to supply the data for the list.
    buddies.setModel(buddiesData);
}

/**
 * Handles updating the user interface when new presence
 * information arrives.
 *
 * @param   presenceInfo the Presence Advertisement containing the
 *           newly arrived presence information.
 */
public void presenceUpdated(PresenceAdvertisement presenceInfo)
{
    String buddyName = null;

    // First check that this buddy is someone we're interested
    // in displaying presence information for.
```

```
buddyName =
    (String) buddyEmailAddresses.get(
        presenceInfo.getEmailAddress());
if (null != buddyName)
{
    // Add the Peer ID to our Hashtable of Peer IDs.
    buddyPeerIDs.put(presenceInfo.getEmailAddress(),
        presenceInfo.getPeerID());

    // Update the list entry.
    for (int i = 0; i < buddiesData.getSize(); i++)
    {
        String currentBuddy = (String) buddiesData.get(i);

        // See if this is the right buddy to update.
        if (currentBuddy.indexOf(buddyName) != -1)
        {
            // Update the buddy name (in case it changed).
            String buddyDisplayName = presenceInfo.getName();
            String buddyEmailAddress =
                presenceInfo.getEmailAddress();
            if (null == buddyDisplayName)
            {
                buddyDisplayName = buddyEmailAddress;
            }

            // Fix hashtables to reflect changes in name.
            buddyNames.remove(buddyName);
            buddyNames.put(buddyDisplayName, buddyEmailAddress);
            buddyEmailAddresses.remove(buddyEmailAddress);
            buddyEmailAddresses.put(buddyEmailAddress,
                buddyDisplayName);

            // Update the list element to reflect the
            // presence change.
            buddyDisplayName += " ";
            switch (presenceInfo.getPresenceStatus())
            {
                case PresenceService.OFFLINE:
                {
                    buddyDisplayName += "(Offline)";
                    break;
                }
```

continues

Listing 11.17 **Continued**

```
                        case PresenceService.ONLINE:
                        {
                            buddyDisplayName += "(Online)";
                            break;
                        }

                        case PresenceService.BUSY:
                        {
                            buddyDisplayName += "(Busy)";
                            break;
                        }

                        case PresenceService.AWAY:
                        {
                            buddyDisplayName += "(Away)";
                            break;
                        }

                        default:
                        {
                            buddyDisplayName += "(Offline)";
                            break;
                        }
                    }
                    buddiesData.set(i, buddyDisplayName);
                }
            }
        }
}

/**
 * Starts the BuddyList application.
 *
 * @param   args the arguments to use to start the application.
 * @return  a status value.
 */
public int startApp(String[] args)
{
    int result = 0;

    try
```

```
{
    // Find the Presence service on the peer group.
    ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
        new URL((PresenceService.refModuleClassID)));
    presence = (PresenceService) peerGroup.lookupService(classID);

    // Register ourselves as a presence listener.
    presence.addListener(this);

    // Find the Chat service on the peer group.
    classID = (ModuleClassID) IDFactory.fromURL(
        new URL((ChatService.refModuleClassID)));
    chat = (ChatService) peerGroup.lookupService(classID);

    // Register ourselves as a chat request listener.
    chat.addListener(chatHandler);

    // Prompt the user to enter his own user information.
    do
    {
        emailAddress = JOptionPane.showInputDialog(null,
            "What is your email address?",
            "Configuration: Step 1 of 2",
            JOptionPane.QUESTION_MESSAGE);
    } while (null == emailAddress);

    do
    {
        displayName = JOptionPane.showInputDialog(null,
            "Enter a display name:",
            "Configuration: Step 2 of 2",
            JOptionPane.QUESTION_MESSAGE);
    } while (null == displayName);

    // Announce that we are online.
    presenceStatus = PresenceService.ONLINE;
    presence.announcePresence(presenceStatus, emailAddress,
        displayName);

    // Show the user interface.
    initializeUserInterface();
    setVisible(true);
}
```

continues

Listing 11.17 **Continued**

```
            catch (Exception e)
            {
                result = 1;
            }

            return result;
        }

        /**
         * Stop the application.
         */
        public void stopApp()
        {
            if (null != presence)
            {
                // Remove ourselves as a presence listener.
                presence.removeListener(this);

                presence = null;
            }

            if (null != chat)
            {
                // Remove ourselves as a chat request listener.
                chat.removeListener(chatHandler);

                chat = null;
            }

            // Hide the user interface.
            setVisible(false);
        }
    }
```

The `BuddyList` class implements the `PresenceListener` interface, allowing it to update the user interface as updated presence information is received. The `BuddyList` class also implements the `ChatListener` interface so that it can ensure that requests to start a chat session are approved only for users who are in the user's list of buddies.

The `BuddyList` class relies on a separate class to handle presenting a user interface for a chat session. The `ChatDialog` class, shown in Listing 11.18, is displayed by `BuddyList` when a chat session has been successfully established.

Listing 11.18 **Source Code for *ChatDialog.java***

```
package com.newriders.jxta.chapter11;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import java.io.IOException;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

import net.jxta.endpoint.Message;

import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;


/**
 * A class to display the chat session user interface and handle the pipes
 * used to send and receive chat messages.
 */
public class ChatDialog extends JFrame implements ActionListener,
    KeyListener
{
```

continues

Listing 11.18 **Continued**

```
/**
 * The text area used to enter a chat message to send to a remote user.
 */
private JTextArea message = new JTextArea(3, 20);

/**
 * The text area to show the incoming and outgoing chat messages in
 * the conversation.
 */
private JTextArea conversation = new JTextArea(12, 20);

/**
 * The input pipe being used to receive chat messages.
 */
private InputPipe inputPipe = null;

/**
 * The output pipe being used to send chat messages.
 */
private OutputPipe outputPipe = null;

/**
 * The name of the remote buddy in the conversation.
 */
private String buddyName = null;

/**
 * The name of the local user in the conversation.
 */
private String displayName = null;

/**
 * The pipe service to use to create Message objects.
 */
private PipeService pipe = null;

/**
 * A thread to handle receiving messages and updating the user
 * interface.
 */
private MessageReader reader = null;
```

```
/**
 * A handler class to deal with closing the window.
 */
public class WindowHandler extends WindowAdapter
{
    /**
     * Handles the window closing.
     *
     * @param   e the object with details of the window event.
     */
    public void windowClosing(WindowEvent e)
    {
        if (reader != null)
        {
            reader.stop();
        }

        setVisible(false);
    }
}

/**
 * A simple thread to handle reading messages from the input pipe and
 * updating the user interface.
 */
public class MessageReader extends Thread
{
    /**
     * The main thread loop.
     */
    public void run()
    {
        while (true)
        {
            try
            {
                Message messageObj = inputPipe.waitForMessage();

                // Make sure that the dialog is visible.
                setVisible(true);

                // Extract the Chat Message.
                StringBuffer chatMessage =
```

continues

Listing 11.18 **Continued**

```
                        new StringBuffer(
                            messageObj.getString("ChatMessage"));

                    // Update the user interface.
                    StringBuffer conversationText =
                        new StringBuffer(conversation.getText());
                    conversationText.append("\n");
                    conversationText.append(buddyName).append("> ");
                    conversationText.append(chatMessage);
                    conversation.setText(conversationText.toString());
                }
                catch (Exception e)
                {
                    System.out.println("Error...: " + e);
                }
            }
        }
    }

    /**
     * Create a new window to handle a conversation with a remote user.
     *
     * @param   buddyName the display name for the remote user in the
     *          chat session.
     * @param   displayName the display name for the local user in the
     *          chat session.
     * @param   pipe the pipe service to use to create messages.
     * @param   inputPipe the pipe to use to receive messages.
     * @param   outputPipe the pipe to use to send messages.
     */
    public ChatDialog(String buddyName, String displayName,
        PipeService pipe, InputPipe inputPipe, OutputPipe outputPipe)
    {
        super();

        this.pipe = pipe;
        this.inputPipe = inputPipe;
        this.outputPipe = outputPipe;
        this.buddyName = buddyName;
        this.displayName = displayName;

        // Initialize the user interface.
```

```
        initializeUserInterface();

        // Set the title of the dialog.
        setTitle("Conversation - " + buddyName);

        reader = new MessageReader();
        reader.start();
    }

    /**
     * Handles the "Send" button.
     *
     * @param   e the event corresponding to the button being pressed.
     */
    public void actionPerformed(ActionEvent e)
    {
        sendMessage();
    }

    /**
     * Initializes the dialog's user interface.
     */
    public void initializeUserInterface()
    {
        Container framePanel = getContentPane();
        JPanel conversationPanel = new JPanel();
        JPanel sendPanel = new JPanel();
        JButton sendButton = new JButton("Send!");
        GridBagLayout layout = new GridBagLayout();
        JScrollPane    messagePane = new JScrollPane(message);

        GridBagConstraints constraints = new GridBagConstraints();

        constraints.gridx = 0;
        constraints.gridy = 0;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.anchor = GridBagConstraints.WEST;
        constraints.weightx = 1;
        constraints.weighty = 1;
        constraints.fill = GridBagConstraints.HORIZONTAL;
        layout.addLayoutComponent(messagePane, constraints);
```

continues

Listing 11.18 **Continued**

```
constraints.gridx = 1;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.anchor = GridBagConstraints.WEST;
constraints.weightx = 0.1;
constraints.weighty = 1;
constraints.fill = GridBagConstraints.BOTH;
layout.addLayoutComponent(sendButton, constraints);

sendPanel.setLayout(layout);
sendPanel.setBorder(BorderFactory.createTitledBorder(
    "Compose A Message:"));
sendPanel.add(messagePane);
sendPanel.add(sendButton);

conversationPanel.setLayout(new BorderLayout());
conversationPanel.setBorder(
    BorderFactory.createTitledBorder("Conversation:"));
conversationPanel.add(new JScrollPane(conversation),
    BorderLayout.CENTER);
conversation.setEditable(false);

constraints.gridx = 0;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.anchor = GridBagConstraints.NORTH;
constraints.weightx = 1;
constraints.weighty = 1;
constraints.fill = GridBagConstraints.BOTH;
layout.addLayoutComponent(conversationPanel, constraints);

constraints.gridx = 0;
constraints.gridy = 1;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.anchor = GridBagConstraints.NORTH;
constraints.weightx = 1;
constraints.weighty = 0;
constraints.fill = GridBagConstraints.BOTH;
layout.addLayoutComponent(sendPanel, constraints);
```

```
        framePanel.setLayout(layout);
        framePanel.add(conversationPanel);
        framePanel.add(sendPanel);

        sendButton.addActionListener(this);
        message.addKeyListener(this);

        pack();
    }

    /**
     * Invoked when a key has been pressed.
     *
     * @param   e the event describing the key event.
     */
    public void keyPressed(KeyEvent e)
    {
        // Do nothing. Only need keyReleased method from KeyListener.
    }

    /**
     * Invoked when a key has been released.
     *
     * @param   e the event describing the key event.
     */
    public void keyReleased(KeyEvent e)
    {
        // Handle the user pressing Return in the message composition
        // text area.
        if (KeyEvent.VK_ENTER == e.getKeyCode())
        {
            sendMessage();
        }
    }

    /**
     * Invoked when a key has been typed.
     *
     * @param   e the event describing the key event.
     */
    public void keyTyped(KeyEvent e)
    {
        // Do nothing. Only need keyReleased method from KeyListener.
```

continues

Listing 11.18 **Continued**

```
    }

    /**
     * Send the message in the message composition text area to the
     * remote user.
     */
    public void sendMessage()
    {
        StringBuffer conversationText =
            new StringBuffer(conversation.getText());
        String messageString = message.getText();

        // Make sure that there is something to send!
        if ((null != messageString) && (0 < messageString.length()))
        {
            // Create a new message object.
            Message messageObj = pipe.createMessage();

            // Send the message using the output pipe.
            messageObj.setString("ChatMessage", messageString);

            // Send the message.
            try
            {
                outputPipe.send(messageObj);
            }
            catch (IOException e2)
            {
                System.out.println("Error sending..." + e2);
            }

            // Update the user interface.
            conversationText.append("\n");
            conversationText.append(displayName).append("> ");
            conversationText.append(messageString);
            conversation.setText(conversationText.toString());
            message.setText("");
        }
    }
}
```

The `ChatDialog` user interface enables a user to send a message to the remote user and view a history of the chat session. The `ChatDialog` class is also responsible for managing the input pipe used to send messages and the output pipe used to receive messages.

The Main Application

Like the `ExampleServiceTest` class in Chapter 10, the `JxtaMessenger` class shown in Listing 11.19 is responsible for generating the necessary advertisements for the Chat and Presence services and the `BuddyList` application, and for creating the peer group.

Listing 11.19 **Source Code for *JxtaMessenger.java***

```
package com.newriders.jxta.chapter11;

import java.util.Hashtable;

import java.net.MalformedURLException;
import java.net.UnknownServiceException;
import java.net.URL;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroupParamAdv;

import net.jxta.impl.protocol.EndpointAdv;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;
```

continues

Listing 11.19 **Continued**

```
import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;

import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.chat.ChatServiceImpl;

import com.newriders.jxta.chapter11.impl.presence.PresenceServiceImpl;

import com.newriders.jxta.chapter11.impl.protocol.PresenceAdv;

import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * The main class responsible for creating the chat peer group and starting
 * the Chat and Presence services and the BuddyList application.
 */
public class JxtaMessenger
{
    /**
     * The Peer Group ID for the chat group.
     */
    private static final String refPeerGroupID =
        "urn:jxta:uuid-68B8A7A691684F9C9E05971D66D78ED602";

    /**
     * The Module Specification ID for the peer group's Module
     * Implementation Advertisement.
     */
    private static final String refPeerGroupSpec =
        "urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000019E2"
        + "DDBBCE5FE4957A139B0ECE8DEB46D06";

    /**
     * The new group created by the application.
     */
    private PeerGroup newGroup = null;
```

```
/**
 * The Net Peer Group for the application.
 */
private PeerGroup netPeerGroup = null;

/**
 * Create the main application class.
 */
public JxtaMessenger()
{
    super();
}

/**
 * Creates a Module Class Advertisement using the given parameters.
 *
 * @param       moduleClassID the Module Class ID for the advertisement.
 * @param       name the symbolic name of the advertisement.
 * @param       description the description of the advertisement.
 * @exception   UnknownServiceException if the moduleClassID string
 *              is malformed.
 * @exception   MalformedURLException if the moduleClassID string
 *              is malformed.
 */
private ModuleClassAdvertisement createModuleClassAdv(
    String moduleClassID, String name, String description)
        throws UnknownServiceException, MalformedURLException
{
    // Create the class ID from the refModuleClassID string.
    ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
        new URL((moduleClassID)));

    // Create the Module Class Advertisement.
    ModuleClassAdvertisement moduleClassAdv =
        (ModuleClassAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleClassAdvertisement.getAdvertisementType());

    // Configure the Module Class Advertisement.
    moduleClassAdv.setDescription(description);
    moduleClassAdv.setModuleClassID(classID);
    moduleClassAdv.setName(name);
```

continues

Listing 11.19 **Continued**

```
        // Return the advertisement to the caller.
        return moduleClassAdv;
    }

    /**
     * Creates a Module Implementation Advertisement for the service using
     * the specification ID in the passed in ModuleSpecAdvertisement
     * advertisement.  Use the given ModuleImplAdvertisement to create the
     * compatibility element of the Module Implementation Advertisement.
     *
     * @param       groupImpl the ModuleImplAdvertisement of the parent
     *              peer group.
     * @param       moduleSpecAdv the source of the specification ID.
     * @param       description of the module implementation.
     * @param       code the fully qualified name of the module
     *              implementation's class.
     * @return      the generated Module Implementation Advertisement.
     */
    private ModuleImplAdvertisement createModuleImplAdv(
        ModuleImplAdvertisement groupImpl,
        ModuleSpecAdvertisement moduleSpecAdv,
        String description, String code)
    {
        // Get the specification ID from the passed advertisement.
        ModuleSpecID specID = moduleSpecAdv.getModuleSpecID();

        // Create the Module Implementation Advertisement.
        ModuleImplAdvertisement moduleImplAdv =
           (ModuleImplAdvertisement) AdvertisementFactory.newAdvertisement(
               ModuleImplAdvertisement.getAdvertisementType());

        // Configure the Module Implementation Advertisement.
        moduleImplAdv.setCode(code);
        moduleImplAdv.setCompat(groupImpl.getCompat());
        moduleImplAdv.setDescription(description);
        moduleImplAdv.setModuleSpecID(specID);
        moduleImplAdv.setProvider("Brendon J. Wilson");

        // Return the advertisement to the caller.
        return moduleImplAdv;
    }
```

```
/**
 * Creates a Module Specification Advertisement using the
 * given parameters.
 *
 * @param        moduleSpecID the Module Specification ID for
 *               the advertisement.
 * @param        name the symbolic name of the advertisement.
 * @param        description the description of the advertisement.
 * @exception    UnknownServiceException if the moduleSpecID string
 *               is malformed.
 * @exception    MalformedURLException if the moduleSpecID string
 *               is malformed.
 */
private ModuleSpecAdvertisement createModuleSpecAdv(String moduleSpecID,
    String name, String description)
        throws UnknownServiceException, MalformedURLException
{
    // Create the specification ID from the refModuleSpecID string.
    ModuleSpecID specID = (ModuleSpecID) IDFactory.fromURL(
        new URL((moduleSpecID)));

    // Create the Module Specification Advertisement.
    ModuleSpecAdvertisement moduleSpecAdv =
        (ModuleSpecAdvertisement) AdvertisementFactory.newAdvertisement(
            ModuleSpecAdvertisement.getAdvertisementType());

    // Configure the Module Specification Advertisement.
    moduleSpecAdv.setCreator("Brendon J. Wilson");
    moduleSpecAdv.setModuleSpecID(specID);
    moduleSpecAdv.setDescription(description);
    moduleSpecAdv.setName(name);
    moduleSpecAdv.setSpecURI(
        "http://www.brendonwilson.com/projects/jxta");
    moduleSpecAdv.setVersion("1.0");

    // Return the advertisement to the caller.
    return moduleSpecAdv;
}

/**
 * Creates a peer group and configures the ChatService and
 * PresenceService implementations to run as peer group services,
 * and configures the BuddyList as a peer group application.
```

continues

Listing 11.19 **Continued**

```
     *
     * @exception   Exception, PeerGroupException if there is a problem
     *              while creating the peer group or the service
     *              advertisements.
     */
    public void createPeerGroup() throws Exception, PeerGroupException
    {
        // The name and description for the peer group.
        String name = "JXTA Messenger Group";
        String description =
            "A peer group for the Chapter 11 example application.";

        // The Discovery service to use to publish the module and peer
        // group advertisements.
        DiscoveryService discovery = netPeerGroup.getDiscoveryService();

        ModuleImplAdvertisement implAdv =
            netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

        // Create the module advertisements for the Presence service.
        ModuleClassAdvertisement presenceClassAdv = createModuleClassAdv(
            PresenceService.refModuleClassID, "Presence Service",
            "A service to provide presence information.");
        ModuleSpecAdvertisement presenceSpecAdv = createModuleSpecAdv(
            PresenceServiceImpl.refModuleSpecID, "Presence Service",
            "A Presence service specification");
        ModuleImplAdvertisement presenceImplAdv = createModuleImplAdv(
            implAdv, presenceSpecAdv,
            "The reference Presence service implementation",
            "com.newriders.jxta.chapter11.impl.presence."
            + "PresenceServiceImpl");

        // Create the module advertisements for the Chat service.
        ModuleClassAdvertisement chatClassAdv = createModuleClassAdv(
            ChatService.refModuleClassID, "Chat Service",
            "A service to provide chat capabilities.");
        ModuleSpecAdvertisement chatSpecAdv = createModuleSpecAdv(
            ChatServiceImpl.refModuleSpecID, "Chat Service",
            "A Chat service specification");
        ModuleImplAdvertisement chatImplAdv = createModuleImplAdv(
            implAdv, chatSpecAdv,
            "The reference Chat service implementation",
```

```
    "com.newriders.jxta.chapter11.impl.chat.ChatServiceImpl");

// Create the module advertisements for the BuddyList application.
ModuleClassAdvertisement appClassAdv = createModuleClassAdv(
    BuddyList.refModuleClassID , "BuddyList Application",
    "An application providing a simple chat application.");
ModuleSpecAdvertisement appSpecAdv = createModuleSpecAdv(
    BuddyList.refModuleSpecID, "BuddyList Application",
    "The BuddyList application specification");
ModuleImplAdvertisement appImplAdv = createModuleImplAdv(
    implAdv, appSpecAdv ,
    "The reference BuddyList application implementation",
    "com.newriders.jxta.chapter11.BuddyList");

// Get the parameters for the peer group's Module Implementation
// Advertisement to which we will add our service.
StdPeerGroupParamAdv params =
    new StdPeerGroupParamAdv(implAdv.getParam());

// Get the services from the parameters.
Hashtable services = params.getServices();

// Add the Chat and Presence services to the set of services.
services.put(presenceClassAdv.getModuleClassID(), presenceImplAdv);
services.put(chatClassAdv.getModuleClassID(), chatImplAdv);

// Set the services on the parameters.
params.setServices(services);

// Replace the applications in the parameters.
Hashtable applications = new Hashtable();

// Add the BuddyList to the applications.
applications.put(appClassAdv.getModuleClassID(), appImplAdv);

// Set the applications on the parameters.
params.setApps(applications);

// Set the parameters on the implementation advertisement.
implAdv.setParam((StructuredDocument) params.getDocument(
    new MimeMediaType("text", "xml")));

// VERY IMPORTANT! You must change the module specification ID for
```

continues

Listing 11.19 **Continued**

```
        // the implementation advertisement. If you don't, the new peer
        // group's module specification ID will still point to the old
        // specification, and the new service will not be loaded.
        implAdv.setModuleSpecID((ModuleSpecID) IDFactory.fromURL(
            new URL(refPeerGroupSpec)));

        // Publish the Presence module class and spec advertisements.
        discovery.publish(presenceClassAdv, DiscoveryService.ADV);
        discovery.remotePublish(presenceClassAdv, DiscoveryService.ADV);
        discovery.publish(presenceSpecAdv, DiscoveryService.ADV);
        discovery.remotePublish(presenceSpecAdv, DiscoveryService.ADV);

        // Publish the Presence module class and spec advertisements.
        discovery.publish(chatClassAdv, DiscoveryService.ADV);
        discovery.remotePublish(chatClassAdv, DiscoveryService.ADV);
        discovery.publish(chatSpecAdv, DiscoveryService.ADV);
        discovery.remotePublish(chatSpecAdv, DiscoveryService.ADV);

        // Publish the Peer Group implementation advertisement.
        discovery.publish(implAdv, DiscoveryService.ADV);
        discovery.remotePublish(implAdv, DiscoveryService.ADV);

        // Create the Peer Group ID.
        PeerGroupID groupID = (PeerGroupID) IDFactory.fromURL(
            new URL((refPeerGroupID)));;

        // Create the new group using the group ID, advertisement, name,
        // and description.
        newGroup = netPeerGroup.newGroup(groupID, implAdv, name,
            description);

        // Need to publish the group remotely only because newGroup()
        // handles publishing to the local peer.
        PeerGroupAdvertisement groupAdv =
            newGroup.getPeerGroupAdvertisement();
        discovery.remotePublish(groupAdv, DiscoveryService.GROUP);

        // Start the peer group's applications.
        newGroup.startApp(null);
    }

    /**
     * Starts the JXTA platform.
```

```
     *
     * @exception   PeerGroupException thrown if the platform can't
     *              be started.
     */
    public void initializeJXTA() throws PeerGroupException
    {
        netPeerGroup = PeerGroupFactory.newNetPeerGroup();
    }

    /**
     * Starts the application.
     * @param args an array of command-line arguments
     */
    public static void main(String[] args)
    {
        JxtaMessenger messenger = new JxtaMessenger();

        try
        {
            // Initialize the JXTA platform.
            messenger.initializeJXTA();

            // Create the group.
            messenger.createPeerGroup();
        }
        catch (Exception e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
    }
}
```

The `JxtaMessenger`'s `createPeerGroup` does the majority of the work, generating the Module Implementation Advertisement for the peer group and populating it with the Module Implementation Advertisements for each of the two services and the application. The required Module Class IDs and Module Specification IDs were generated using the `GenerateID` application developed in Chapter 10 and were stored as static variables in `ChatService`, `ChatServiceImpl`, `PresenceService`, `PresenceServiceImpl`, and `BuddyList` for convenience.

Running JXTA Messenger

After all the classes have been compiled, you should be able to run the JXTA Messenger application from a directory containing the JXTA JARs using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter11.JxtaMessenger
```

To see the JXTA Messenger in action, you will probably want to start a second instance of `JxtaMessenger` from a different directory using different TCP and HTTP ports. This will enable you to experiment with the application on your own machine.

After the peer group has been created and started by `JxtaMessenger`, the `BuddyList` application prompts you to enter an email address and display a name to use when sending presence information to other peers. This information will also be used when requesting or approving a chat session. After you have entered this information, the main user interface should appear as shown in Figure 11.1.

Figure 11.1 The main JXTA Messenger user interface.

By default, your presence status will be set to Online. You can change your presence status from the My Status menu item under the Actions menu, as shown in Figure 11.2.

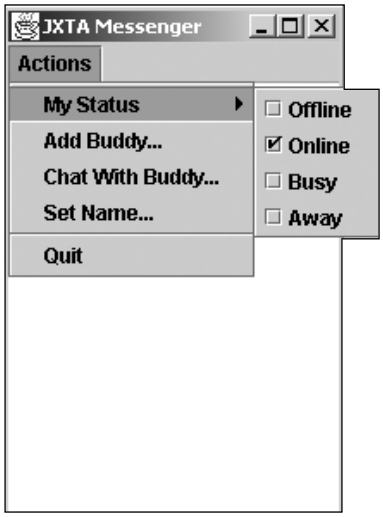

Figure 11.2 Setting your presence status.

When you set your status, a Presence Advertisement is published both locally and remotely using the peer group's Presence service.

To monitor other users' presence status, you need to add the user to your list of buddies using the Add Buddy menu item under the Actions menu. You will be prompted for an email address that the `BuddyList` application can use to search for a Presence Advertisement using the Presence service. The buddy will be added to the list, but the buddy's status will remain as Offline until `BuddyList`'s `presenceUpdated` method is called by the `PresenceService` instance to notify `BuddyList` that a Presence Advertisement has been found for the buddy.

When presence information is received, the `BuddyList` instance updates both the user interface and its internal set of buddies. The Peer ID associated with the buddy is also stored to allow the user to send chat requests to the buddy without using propagation.

To chat with a buddy, click a buddy in the list and select the Chat with Buddy menu item under the Actions menu. The `BuddyList` class uses the Chat service to send an Initiate Chat Request Message to the remote user. When the `BuddyList` class receives a request to initiate a chat session, it checks whether the requesting buddy is in the user's list of buddies. If it is, the `BuddyList` class creates a pipe using the `BidirectionalPipeService` and sends an Initiate Chat Response Message. Otherwise, the `BuddyList` class ignores the request. This means that only users who are in each other's buddy list can chat.

When the `BuddyList` receives notification that the request to chat has been approved, it attempts to connect to the remote peer using the Pipe Advertisement contained in the response. If this connection is successful, the `BuddyList` spawns a `ChatDialog` to manage the chat session, as shown in Figure 11.3.

Figure 11.3 The `ChatDialog` user interface.

`ChatDialog` uses the `OutputPipe` bound by the `BidirectionalPipeService` to send messages entered by the user. Messages received on the `InputPipe` bound by the `BidirectionalPipeService` are added to the text area showing the conversation history. When the user closes the dialog box, the pipes are closed and no further communication is possible without requesting a chat session using the same procedure as before.

Summary

Although the chat application developed in this chapter might not be as fully featured as MSN Messenger or ICQ, it still demonstrates how the JXTA protocols covered in this book can be used to create a complete P2P solution:

- **The Peer Discovery Protocol**—The Presence service implementation uses the Discovery service to search for Presence Advertisements containing presence information for a user. The Discovery service is also used by the Presence service implementation to discover presence information for a user.

- **The Resolver Protocol**—The Chat service implementation uses the Resolver service to handle sending and receiving messages used to request and approve a request to establish a chat session.
- **The Rendezvous Protocol**—Although it is not used directly, the Rendezvous service provides the Discovery service with the capability to publish Presence Advertisements to many peers.
- **The Pipe Binding Protocol**—The Pipe Binding Protocol is used by the `BidirectionPipeService`'s use of the Pipe service to establish a two-way communication channel to a remote peer for conducting the chat session.
- **The Endpoint Routing Protocol**—Underneath all the protocols, the Endpoint Routing Protocol provides the mechanism required to route messages to destination peers.

The only protocol not used by this application is the Peer Information Protocol, which, given its current state, is forgivable. The sample application also re-enforces the demonstration of peer groups, modules, and services given in Chapter 10.

So where do you go from here? The next chapter examines some of the more ambitious projects currently being built by the JXTA Community. The next chapter also provides information on how you can get involved with Project JXTA, contribute to an existing project, or propose a project of your own.