RECURSION SOFTWARE, Inc.

Software for Software Developers™

Developing Peer Applications with Voyager

By Thomas Wheeler

Introduction

Background

Over the past few years, innovative collaborative technologies have appeared enabling people to join together and participate in new ways. Distributed computing projects such as SETI@home have empowered users to donate the unused computing cycles of their own systems to work on computationally large problems. Instant messaging services allow millions of people to communicate and collaborate in real time. Peer-to-peer (P2P) file-sharing, through applications like Gnutella and Freenet, has offered a radically new mechanism for Internet users to find and share files directly with each other, often without requiring a central authority or server. These are specific, narrow examples of P2P technology. True P2P computing requires a platform that makes it easy to build a wide network of distributed services and applications in which every device is addressable as a peer and peers can easily and dynamically communicate and interoperate.

Peer-to-Peer Solutions

The acceptance of first-generation peer-to-peer applications has brought peer-to-peer computing high visibility. The success of these applications is testimony to the power and viability of the peer-to-peer model. Peer-to-peer computing is generating excitement because it offers an intuitive, simple model for fundamental computing activities: *discovery, searching, and sharing of resources*. Although today's P2P applications are relatively primitive, they hint at what will be possible in the future. Peer-to-peer computing enables applications that are collaborative and communication-focused; more probabilistic than deterministic. Applications that are well-suited to this model are those which can tolerate the coming and going of individual peers. In this model, the failure of a single peer does not result in a failure in the entire system, as is often the case with today's distributed applications. The information exchanged in P2P environments can be timely and accurate, yet results may vary from time to time depending on which peer group members are available. High availability comes through the existence of many peers in a group, making it probable that at any time there is at least one peer in the group able to satisfy a request. This stands in stark contrast to computing models in use today, where high availability comes through complex load-balancing and application fail-over mechanisms. Peer-to-peer computing leverages available computing performance, storage, and bandwidth found on systems around the globe. It works because people realize that there is value in sharing what they have with others to reap the benefits of what others have to share.

Peer-to-Peer Advantages

Existing distributed applications follow the client-server architecture. Web applications, n-tier applications, and Web Services applications all share this model. For many future applications, this will continue to be a successful

architecture. However, for some applications a P2P architecture provides advantages that are unique or difficult to implement using traditional technologies. The advantages of a P2P architecture include:

- *Reliability:* Client-server applications are only as reliable as the server (or server cluster) hosting the application. P2P systems provide reliability through the replication of services, computing power, and storage. This redundancy provides reliability that cannot be easily achieved in a client-server deployment.
- *Utilization:* Traditional client-server applications are bounded by the resources of the server (or cluster). Under-utilization is common: computing power, memory and long-term storage, and services are used to only a fraction of their potential. Over-utilization occurs when the server's resources are insufficient to meet the demand. (One famous example is the "Slashdot effect".) In a P2P architecture, each node acts as both a consumer and a supplier or resources. This minimizes both under-utilization and over-utilization.
- *Efficiency:* Efficiency is related to utilization. Client-server applications typically result in poor efficiency. Servers are required to have excess capacity sufficient to meet the highest reasonably expected load, but this level of load is rare. In a P2P environment, each node can contribute its "spare" resources to the group, increasing efficiency and utilization.
- *Manageability:* A typical client-server system requires dedicated staff with expertise in a variety of specialized skills to ensure availability of the server application and to perform administrative functions. A P2P system is essentially self-managing: nodes join and leave the network dynamically, but the resources of the system are always available.
- *Control:* Client-server architectures implicitly require centralized control of resources. In a P2P system, each node can be controlled by the individual or group which owns the node. This leads to increased efficiencies, and a sense of empowerment for the individuals involved.
- *Networking:* A client-server system requires the server to provide a high-bandwidth, high-reliability networking infrastructure. In contrast, a P2P architecture provides bandwidth and reliability through quantity – the number of peers available. In addition, a P2P architecture can reduce bandwidth by allowing peers to communicate directly rather than through a centralized (possibly distant) server. Finally, P2P architectures support a dynamic, flexible network architecture.

The business advantages of P2P computing are directly related to the architectural advantages: increasing the utilization of existing computing resources, increasing the availability of services, and decreasing the cost of managing systems has a direct positive impact on the bottom line.

Voyager: A Distributed Computing Framework

Voyager provides a feature-rich framework that supports the rapid development of distributed computing applications, including P2P applications, on any Java-enabled platform. Voyager's patented technology builds on existing ideas and key concepts that make Java-based distributed computing powerful and flexible.

Voyager's core provides an infrastructure and framework. This infrastructure includes the management of network connections, threads, and runnable tasks; a dynamic proxy class generator; remote messaging with support for multiple messaging systems; remote class-loading; multi-home support; remote naming and lookup services; and distributed garbage collection. This core set of functionality provides the backbone which supports an unmatched set of advanced features including: mobility and agents; multicast, publish/subscribe, one-way, and future messages; and replicating and load balancing naming services.

Core Voyager Features for P2P

A P2P application requires a core set of infrastructure services. Voyager provides the core functionality required for P2P services and applications. These services include:

- *Resource publication:* Peers must make themselves and their services known to the network in order to be utilized by other peers.
- *Resource location:* Before utilizing a resource, a peer must locate it on the network. A mechanism for resource discovery must be in place to allow this.
- *Resource utilization:* Once a resource is discovered, it can be utilized. This implies sending messages to a remote service.

Resource publication can be accomplished through the Voyager Naming Service. By binding a service into the VNS, we make it available to the network. But how do we make it available to peers?

Voyager's *Space* feature provides the infrastructure necessary for service discovery. A Voyager Space is a distributed, dynamically linked group of *Subspaces* for broadcasting and receiving messages. Think of a Space as a peer group, and a Subspace as a peer which sends and receives *messages*. Messages are application-specific events or objects which are broadcast to each member of a Space. The connections between Subspaces form the "network topology" of the Space. Each connection is bidirectional, allowing Subspaces (peers) to simultaneously publish messages and receive them from a neighboring Subspace. A Voyager application can create or join any number of Spaces (peer groups).

Initial discovery of a peer group is performed through Voyager's naming service. This simplifies and standardizes the process of initial lookup.

Resource utilization can be accomplished through direct remote invocation of the service. Voyager supports multiple types of remote invocations, including one-way, synchronous, and future (asynchronous).

The following sections will demonstrate how Voyager can be used to rapidly and easily develop P2P applications.

Tutorial

Developing a Peer-to-Peer Chat Application with Voyager

The standard introductory application for Peer-to-Peer systems is a chat application. It is simple, well-understood, and well-suited for a P2P implementation. The application we will develop here demonstrates the ease with which P2P applications can be implemented with Voyager. To build this application, you will need to have the latest version of Voyager installed and properly configured. If you haven't done this, please go to http://www.recursionsw.com and do so now.

Features and Limitations

The chat application we will develop will demonstrate core P2P features. It will include the ability to join a Space (peer group); view, create, and join chatrooms; and send and receive messages to/from each joined chatroom. Our use cases are as follows:

- *Connect*. The user must first connect to the peer group by providing one or more candidate peer addresses to attempt to connect to.
- *Login*. The user must log in by entering their username ("nickname") and joining the group. The provided name must be unique within the group.
- *View chatrooms*. The user may obtain a list of all known chatrooms.
- *Join chatroom.* The user may join an existing chatroom. The existing members of the chatroom will be notified that the user has joined.
- *Create chatroom.* The user may create a new chatroom. All users will be notified of the creation of the new chatroom.

- *Leave chatroom.* The user may leave a chatroom. The existing members of the chatroom will be notified that the user has left.
- *Send message.* The user may send a message to a chatroom they have joined. All members of the chatroom will receive the message sent.
- *Send private message.* The user may send a message directly to another user.
- *Exit.* The user may exit the chat application. Members of chatrooms they have joined will be notified the user has left the chatroom.

This application is intended as a tutorial sufficient to give the beginning Voyager user insight into how to create P2P applications with Voyager. It explicitly does not include advanced features or the sophisticated error handling required to make an application robust. However, parts of the code that can be expanded on will be noted, along with suggestions for improvements.

Application Architecture

The application is divided into two architectural layers: a framework of P2P classes, and the application layer. The P2P framework is responsible for low-level communications, interfacing directly with Voyager classes to communicate. The application layer is largely Voyager-independent.

The deployment model consists of at least one server and at least one peer. Within an individual network, there will be one server and 1..n peers. The server provides management of application-wide state: nicknames and chatroom names. Peers communicate with each other through messages or directly through remote invocations.

Development – Core Classes

Voyager provides a core infrastructure for building networked applications. To help build our P2P chat application, we will create several core classes that will simplify development. These are built on top of Voyager's core APIs and provide P2P abstractions that will be useful as we develop application classes.

Peer

A Peer represents a single node in the P2P network. It has a connection to the network through a PeerConnection, and uses a MessageDispatcher to dispatch incoming messages from the network.

```
/**
 * A Peer represents the local endpoint in a peer network.  It is connected to
 * the network via a PeerConnection.  Messages can be broadcast to the network
 * through this connection.
 *
 */
public class Peer
{
  private PeerConnection peerConnection;
  private MessageDispatcher dispatcher;

  /**
   * Create a new peer.
   * @param remoteSubspace The remote subspace to connect to.
   * @throws PeerException There was a problem creating the peer.
   */
  public Peer(ISubspace remoteSubspace) throws PeerException
  {
    init( createLocalSubspace(null), remoteSubspace );
  }

  /**
```

```
     * Create a new peer.
     * @param remoteCandidates A list of remote addresses of candidate peers to connect to.
     * @throws PeerException There was a problem creating the peer.
     */
    public Peer(String bindName, String[] remoteCandidates) throws PeerException
    {
      ISubspace localSubspace = createLocalSubspace(bindName);

      ISubspace remoteSubspace = findRemoteSubspace(bindName, remoteCandidates);

      init( localSubspace, remoteSubspace );
    }

    private void init( ISubspace localSubspace, ISubspace remoteSubspace ) throws
  PeerException
    {
      if( remoteSubspace != null )
      {
        localSubspace.connect(remoteSubspace);
      }

      dispatcher = new MessageDispatcher(this);
      peerConnection = new PeerConnection(this, localSubspace, remoteSubspace);
    }

    /**
     * Get the local peer address.
     * @return The local address.
     */
    public String getLocalAddress()
    {
      return peerConnection.getLocalAddress();
    }

    /**
     * Get the remote peer address.
     * @return The remote address.
     */
    public String getRemoteAddress()
    {
      return peerConnection.getRemoteAddress();
    }

    /**
     * Get the peer connection used to broadcast messages.
     * @return The connection.
     */
    public PeerConnection getPeerConnection()
    {
      return peerConnection;
    }

    /**
     * Get the message dispatcher used to dispatch messages and register
     * message handlers.
     * @return
     */
    public MessageDispatcher getMessageDispatcher()
    {
      return dispatcher;
    }

    /**
     * Disconnect from the peer network.
     */
    public void disconnect()
    {
      peerConnection.disconnect();
```

```
  }

  /**
   * Create the local subspace.
   * @throws PeerException Could not create the local subspace.
   */
  private ISubspace createLocalSubspace(String bindName) throws PeerException
  {
    ISubspace localPeer;
    Proxy p = null;
    try
    {
      localPeer = new Subspace();
      localPeer.setPurgePolicy(Subspace.ALL);
      try
      {
        p = Proxy.export(localPeer, "10102");
      }
      catch( Exception e )
      {
        p = Proxy.export(localPeer);
      }
      if( bindName != null )
        Namespace.bind(bindName, p);
    }
    catch (ExportException e1)
    {
      throw new PeerException(
        "Unable to start local peer (export exception: "
          + e1.getMessage()
          + ")");
    }
    catch (NamespaceException e2)
    {
      throw new PeerException(
        "Unable to start local peer (naming exception: "
          + e2.getMessage()
          + ")");
    }
    return (ISubspace) p;
  }

  /**
   * Attempt to connect to one of the candidate remote peers.
   * @param remotePeers A list of candidate remote peers.
   */
  public ISubspace findRemoteSubspace(String bindName, String[] remotePeers)
  {
    ISubspace firstPeer = null;

    // Discover a remote peer
    for( int idx = 0; idx < remotePeers.length && firstPeer == null; idx++)
    {
      String address = (String) remotePeers[idx];
      try
      {
        firstPeer = (ISubspace) Namespace.lookup(address + bindName);
      }
      catch (Exception e)
      {
        // failure to connect to a peer is not an application failure
      }
    }
    return firstPeer;
  }
}
```

PeerConnection

A PeerConnection represents the local peer's connection to the P2P network. It is implemented through a local Subspace instance connected to a remote peer Subspace. The PeerConnection uses the Subspace to broadcast Messages to other peers and to receive Messages from other peers. Received messages are passed to the MessageDispatcher for handling.

In our chat application, many commands will use the PeerConnection to broadcast a message to the peer network.

```
/**
 * A PeerConnection represents the connection to the peer network.
 * Its primary function is to broadcast messages to the network and dispatch
 * messages received from the network.
 *
 */
public class PeerConnection implements PublishedEventListener
{
  private String localAddress;
  private String remoteAddress;
  private ISubspace localPeer;
  private ISubspace remotePeer;
  private boolean isAlive = true;
  private Peer peer;

  /**
   * Establish a local subspace and connect it to the network (a remote subspace).
   */
  public PeerConnection(Peer peer, ISubspace local, ISubspace remote) throws PeerException
  {
    XURL url;

    this.peer = peer;
    this.localPeer = local;
    this.remotePeer = remote;

    Proxy p = (Proxy) localPeer;
    url = XURL.acquireXURL(p.getURL());
    localAddress = "//" + url.getHost() + ":" + url.getPort();

    if( remotePeer != null )
    {
      url = XURL.acquireXURL(Proxy.of(remotePeer).getURL());
      remoteAddress = "//" + url.getHost() + ":" + url.getPort();
    }

    localPeer.add(this);
  }

  /**
   * Get the peer for this connection.
   * @return The peer.
   */
  public Peer getPeer()
  {
    return peer;
  }

  /**
   * Get the local address of the peer.
   * @return The address.
   */
  public String getLocalAddress()
  {
    return localAddress;
  }
```

```
  /**
   * Get the address of the address of the peer we are connected to.
   * @return The address.
   */
  public String getRemoteAddress()
  {
    return remoteAddress;
  }

  /**
   * Broadcast a new message to the peer network.
   * @param message The message to broadcast.
   * @param targetPeer The address of the single peer intended to handle the message
   * (or null for all peers).
   */
  public void broadcast(Message message, String targetPeer)
  {
    if( isAlive )
    {
      MessageWrapper wrapper = new MessageWrapper(getLocalAddress(), targetPeer, message );
      Publish.invoke( localPeer, wrapper, new Topic( "com.recursionsw.p2p.message" ));
    }
  }

  /**
   * Broad cast a new message to all peers.
   * @param message The message to broadcast.
   */
  public void broadcast(Message message)
  {
    broadcast(message, null);
  }

  /**
   * Receive a message and process it.  If the message has a target address,
   * handle it if we are the target.
   */
  public void publishedEvent(EventObject event, Topic topic)
  {
    if (topic.toString().equals("com.recursionsw.p2p.message"))
    {
      MessageWrapper wrapper = (MessageWrapper) event;
      String target = wrapper.getTargetPeerAddress();
      if (target == null || target.equals(localAddress))
      {
        try
        {
          peer.getMessageDispatcher().dispatch(wrapper);
        }
        catch (Exception e)
        {
          // All application exceptions should be handled by the message dispatcher.
        }
      }
    }
  }

  /**
   * Disconnect our local subspace from the remote peer's subspace.
   */
  public void disconnect()
  {
    isAlive = false;
    localPeer.disconnect(remotePeer);
  }
}
```

Message

A Message is a serializable object, with a name and parameters. Messages are broadcast to each peer through a PeerConnection and handled by a MessageHandler. MessageHandlers are associated with Message names. In essence, think of the Message's name as a method name, or type, and the MessageHandler as the implementation for that method.

There are two ways of creating application-specific Messages. One way is to use the Message class as-is, supplying the constructor with appropriate names and creating appropriate MessageHandler classes to handle each type of message. The second way is to extend the Message class and provide custom behavior through subclasses. Our chat application will use the latter style, creating a subclass of Message called ChatMessage.

```
/**
 * A Message is a serializable object with a name and arguments.  Messages are
 * broadcast to each peer through the local peer's PeerConnection.  When a message
 * is received, its name is used to look up a MessageHandler, which is then
 * called to handle the message.
 *
 */
public class Message implements Serializable
{
  private static long nextId;

  private String name;
  private Object[] arguments;
  private long id;

  /**
   * Create a message from the given name and arguments.
   * @param name The name of the message
   * @param arguments Arguments for the message
   */
  public Message(String name, Object[] arguments )
  {
    this.name = name;
    this.arguments = arguments;
    this.id = getNextid();
  }

  /**
   * Get the message id.
   * @return The id.
   */
  public long getId()
  {
    return id;
  }

  /**
   * Get the arguments to the message.
   * @return The array of arguments
   */
  public Object[] getArguments()
  {
    return arguments;
  }

  /**
   * Get the name of this message
   * @return The message name
   */
  public String getName()
```

```
  {
    return name;
  }

  private synchronized static long getNextid()
  {
    return nextId++;
  }
}
```

MessageHandler

The MessageHandler interface is implemented to provide message handling for a particular type (name) of Message. When a Message arrives at a peer, its name is used to look up a MessageHandler. The MessageHandler is then called to process the message. Our chat application will create a single MessageHandler implementation which will be registered and used to handle all chat messages. Subclasses of ChatMessage will be used to define custom behavior.

```
/**
 * A MessageHandler implements application-specific behavior invoked
 * when the peer receives a message with a name associated with the
 * MessageHandler (as registered in the MessageDispatcher).
 *
 */
public interface MessageHandler
{
  /**
   * Handle the specified message.
   * @param message The message to handle.
   * @throws PeerException There was a problem handling the message.
   */
  void handleMessage( Message message ) throws PeerException;
}
```

MessageDispatcher

The MessageDispatcher handles message processing at each peer. When a message arrives, the MessageDispatcher looks up the MessageHandler based on the message's name and then dispatches the message to that MessageHandler.

If no MessageHandler exists for a message, an exception is raised and returned as the response.

```
/**
 * A MessageDispatcher is responsible for dispatching messages received from other
 * peers, and for managing MessageHandlers.  When a message arrives at a peer,
 * the MessageDispatcher uses its name to look up its associated MessageHandler,
 * and then dispatches the message to that MessageHandler.
 *
 */
public class MessageDispatcher
{
  private static final MessageHandler unknownMessageHandler = new UnknownMessageHandler();
  private Peer peer;
  private Map messageHandlers;

  /**
   * Create a new MessageDispatcher.
   * @param peer The local peer instance.
   */
  public MessageDispatcher(Peer peer)
  {
```

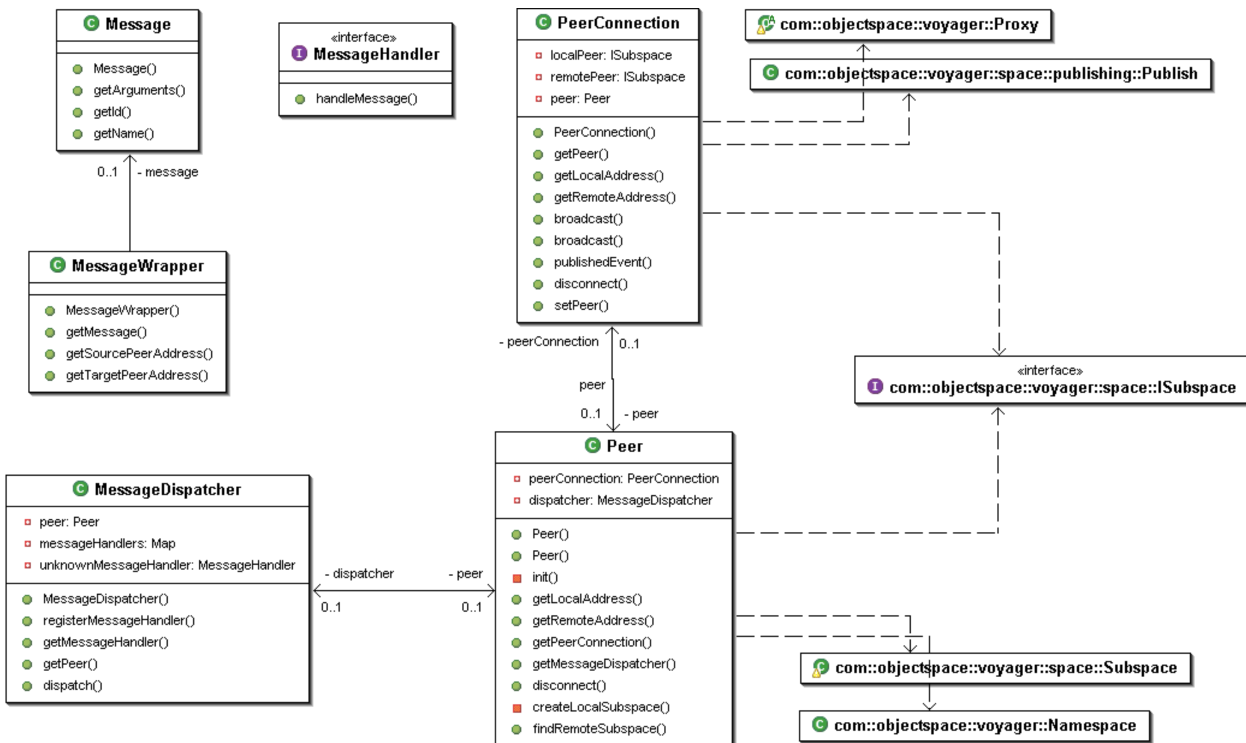
```
    this.peer = peer;
    this.messageHandlers = new HashMap();
  }

  /**
   * Register a MessageHandler for the given message name.  All messages
   * with this name are handled by this handler when they arrive at this peer.
   * @param message The message name.
   * @param handler The message handler.
   */
  public void registerMessageHandler( String message, MessageHandler handler )
  {
    messageHandlers.put(message, handler);
  }

  /**
   * Get the MessageHandler associated with the name.  If no known
   * handler is found, return the UnknownMessageHandler (which throws
   * an exception when its handleMessage() method is called).
   * @param messageName The name of the message.
   * @return The MessageHandler for the message, or UnknownMessageHandler.
   */
  public MessageHandler getMessageHandler( String messageName )
  {
    MessageHandler handler = (MessageHandler) messageHandlers.get( messageName );
    if( handler == null )
      handler = unknownMessageHandler;
    return handler;
  }

  /**
   * Get the peer for this MessageDispatcher.
   * @return The peer.
   */
  public Peer getPeer()
  {
    return peer;
  }

  /**
   * Dispatch a message.  Looks up the MessageHandler for the message name
   * and calls its handleMessage() method to handle the message.
   * @param wrapper The MessageWrapper containing the message.
   * @throws PeerException No MessageHandler was found, or the handleMessage()
   * call threw a PeerException.
   */
  public void dispatch(MessageWrapper wrapper) throws PeerException
  {
    Message message = wrapper.getMessage();
    String messageName = message.getName();
    MessageHandler handler = getMessageHandler( messageName );
    handler.handleMessage(message);
  }
}

/**
 * UnknownMessageHandler throws an exception when its handleMessage()
 * method is called.
 *
 */
class UnknownMessageHandler implements MessageHandler
{
  public void handleMessage(Message message) throws PeerException
  {
    throw new PeerException( "Unknown message type (name=" + message.getName() +")" );
  }
}
```

Following are class and sequence diagrams of the core p2p API classes:

Class Diagram – com.recursionsw.p2p

This diagram shows the major classes and their dependencies. Most of the Voyager dependencies in the application are limited to classes in this package.

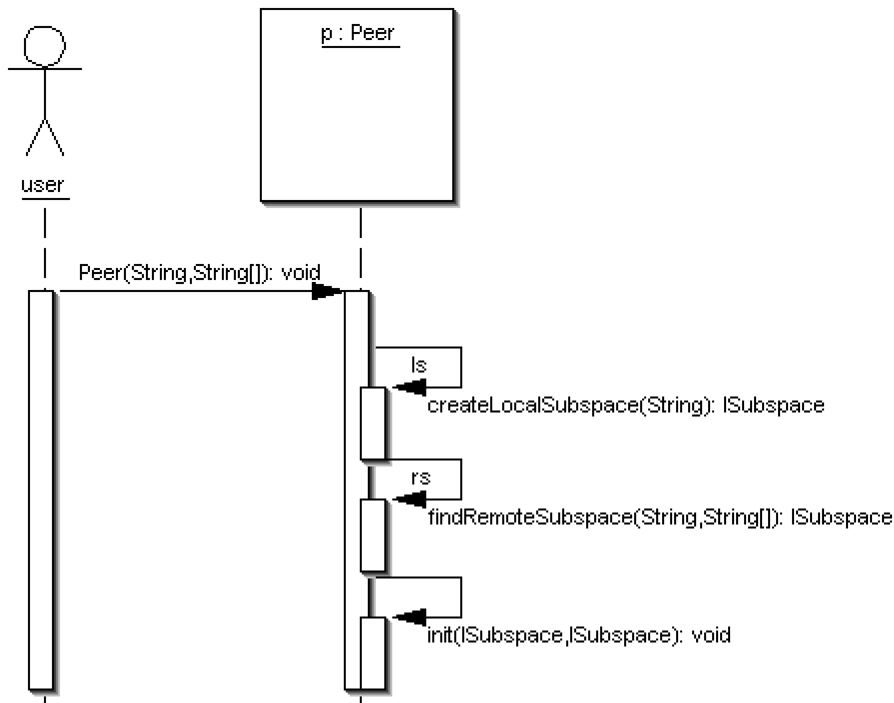
Figure 1. Major Classes



Sequence Diagram – com.recursionsw.p2p

This diagram shows how the peer discovers and connects to a remote peer.

Figure 2. Connection to Remote Peer



Development – Chat Application Classes

As mentioned earlier, the chat application classes depend on the core classes described above. Now that we have developed these P2P classes, we can focus on the development of the chat application.

Our chat application will be split among several packages:

`com.recursionsw.chat` will contain the main application classes and domain classes.
`com.recursionsw.chat.command` will contain core classes and interfaces for the Command pattern.
`com.recursionsw.chat.command.impl` will contain implementation classes for chat commands.
`com.recursionsw.chat.message` will contain core classes and interfaces for chat messages.
`com.recursionsw.chat.message.impl` will contain application-specific message classes.

ChatPeer

Our application class is ChatPeer. It handles application startup, input of commands from the user and their dispatch, and manages application information.

When the ChatPeer application is started, it creates a Peer to connect to the peer network. As input, it accepts a list of candidate addresses of remote peers. If it is unable to connect to any of the candidates, it assumes that it is the only peer in the network and starts a ChatServer. Otherwise, it obtains a ChatServer from the peer it connected to. Finally, it registers a ChatMessageHandler to handle chat application messages broadcast to the peer network.

The ChatPeer uses the command pattern for processing user commands. Commands are entered on the console and output is printed to the console. A Swing-based interface could be developed with relatively little modification to existing code.

```
/**
 * The application class.  ChatPeer handles application startup, command and message
handling,
 * and the input loop.
 */
public class ChatPeer implements Runnable, IChatPeer
{
  private BufferedReader inputReader;
  private String name = null;
  private CommandFactory commandFactory;
  private Peer peer;
  private IChatServer chatServer;
  private Map cachedClients = new HashMap();
  private Map joinedRooms = new HashMap();
  private boolean enteringCommand = false;

  //-----------------
  //Constructors
  //-----------------

  /**
   * Create the application.
   * @param remoteCandidates A list of addresses of remote peer candidates.
   * @throws ChatException
   */
  public ChatPeer(String[] remoteCandidates) throws ChatException
  {
    inputReader = new BufferedReader(new InputStreamReader(System.in));
    commandFactory = new CommandFactory(this);
    IChatPeer remotePeer = null;

    try
    {
      this.peer = new Peer("/PeerSubspace", remoteCandidates);
      peer.getMessageDispatcher().registerMessageHandler("com.recursionsw.chat.message", new
ChatMessageHandler( this ));
      String remoteAddress = peer.getRemoteAddress();
      if( remoteAddress != null )
      {
        remotePeer = (IChatPeer) Namespace.lookup( remoteAddress + "/ChatPeer" );
        chatServer = remotePeer.getServer();
      }
      if( chatServer == null )
      {
        chatServer = new ChatServer(peer);
      }
      Namespace.bind( "/ChatPeer", this );
    }
    catch( Exception e )
    {
      throw new ChatException( e.getMessage() );
    }

    if (name == null)
      name = "?";
```

```
}

//-----------------
//Getter/setter
//-----------------

/**
 * Get the user name.
 * @return The user name.
 */
public String getName()
{
  return name;
}

/**
 * Set the user name.
 * @param newName The new name.
 */
public void setName(String newName)
{
  this.name = newName;
}

/**
 * Get the address of the local peer.
 * @return The address.
 */
public String getLocalAddress()
{
  return peer.getLocalAddress();
}

/**
 * Get the local peer instance.
 * @return The local peer.
 */
public Peer getPeer()
{
  return peer;
}

// ----------------
// IChatPeer
// ----------------

/**
 * Get the chat server.
 * @return The chat server.
 */
public IChatServer getServer()
{
  return chatServer;
}

/**
 * Print a message to the console.  Called by remote peers.
 */
public void message(String from, String message)
{
  print( from + ": " + message );
}

//----------------
//Runnable
//----------------

/**
```

```
    * Input loop: input a command and dispatch it.
    */
   public void run()
   {
     while (true)
     {
       enteringCommand = true;
       String command = inputCommand();
       enteringCommand = false;
       runCommand(command);
     }
   }

   // -----------------
   // Room commands
   // -----------------

   /**
    * Join a room.  Get the room and join it.  Cache the name:room in
 <code>joinedRooms</code>.
    * @param room The name of the room to join.
    */
   public void joinRoom(String room) throws ChatException
   {
     if( !isMemberOf( room ))
     {
       IChatRoom chatRoom = chatServer.getRoom(room);
       chatRoom.join(getName());
       joinedRooms.put(room, chatRoom);
     }
     else
       throw new ChatException( "You are already in room " + room );
   }

   /**
    * Determine if we are a member of the given room.
    * @param room The room to look in.
    * @return true if we are a member, false otherwise.
    */
   public boolean isMemberOf( String room )
   {
     Iterator i = joinedRooms.keySet().iterator();
     while( i.hasNext() )
     {
       String s = (String) i.next();
       if( s.equals(room))
         return true;
     }
     return false;
   }

   /**
    * Leave a room we are a member of.
    * @param name The name of the room to leave.
    * @throws ChatException
    */
   public void leaveRoom(String name) throws ChatException
   {
     IChatRoom chatRoom = (IChatRoom) joinedRooms.get(name);
     if( chatRoom != null )
     {
       joinedRooms.remove(name);
       chatRoom.leave(getName());
     }
     else
       throw new ChatException( "You are not in room " + name );
   }
```

```
//--------------------
//Utility methods
//--------------------

/**
 * Start the application.  Creates a new thread for the input loop.
 */
public void start()
{
  new Thread(this).start();
}

/**
 * Get a map of remote clients we know about.
 * @return The map.
 */
public Map getCachedClients()
{
  return cachedClients;
}

/**
 * Print a message to the console.  If the user is in the process of entering
 * a command, print the prompt after the message; otherwise just print
 * the message.
 * @param message The message to print.
 */
public void print(String message)
{
  if(enteringCommand)
    System.out.print("\n" + message + "\n" + getName() + ":" );
  else
    System.out.println(message);
}

/**
 * Get a command from the console (stdin).
 * @return The command.
 */
private String inputCommand()
{
  try
  {
    System.out.print(name + ":");
    return inputReader.readLine();
  }
  catch (IOException e)
  {
  }
  return "stop";
}

/**
 * Run a command entered on the console.
 * @param command
 */
private void runCommand(String command)
{
  ChatCommand chatCommand = null;
  try
  {
    chatCommand = commandFactory.create(command);
    if (chatCommand != null)
    {
      chatCommand.execute(this);
      return;
    }
    else
```

```
        {
          System.out.println( "I don't understand '" + command + "'" );
        }
      }
      catch (ChatException e)
      {
        System.out.println( e.getMessage() );
      }
    }

    //-----------------
    // Main
    //-----------------

    /**
     * Startup.
     * @param args Commandline arguments.
     */
    public static void main(String[] args)
    {
      String[] remoteCandidates = args;
      try
      {
        Voyager.startup();
        new ChatPeer(remoteCandidates).start();
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }

  }
```

ChatServer

The ChatServer provides a management interface to prevent duplicate sign-ons and duplicate chatrooms, and lookup abilities for peers and chatrooms.

```
/**
 * A ChatServer manages the master list of rooms and peers.
 * @author twheeler
 */
public class ChatServer implements IChatServer
{
  private Map rooms;
  private Peer peer;
  private Map peers;

  /**
   * Create a new chat server.
   * @param peer Local peer instance.
   */
  public ChatServer(Peer peer)
  {
    this.peer = peer;
    rooms = new HashMap();
    rooms.put("Global", new ChatRoom("Global"));

    peers = new HashMap();
  }

  /**
   * Login (verify another peer isn't already connected).
   * @param name The nickname of the client.
```

```
 * @param chatPeer the peer.
 */
public void login(String name, IChatPeer chatPeer) throws ChatException
{
  if( peers.get(name) != null )
    throw new ChatException( "Someone by that name is logged in already" );
  peers.put( name, chatPeer );
  peer.getPeerConnection().broadcast(new LoginMessage(name));
}

/**
 * Leave the peer network.
 * @param name The nickname of the peer logging out.
 */
public void logout(String name)
{
  peers.remove(name);
  peer.getPeerConnection().broadcast(new LogoutMessage(name));
}

/**
 * Look for a particular peer by nickname.
 */
public IChatPeer getPeer(String name)
{
  return (IChatPeer) peers.get(name);
}

/**
 * Change the nickname of a connected peer.
 * @param oldName the original name.
 * @param newName the new name.
 */
public void setName(String oldName, String newName ) throws ChatException
{
  if( peers.get(newName) != null )
    throw new ChatException( "Cannot change name to an existing name" );
  IChatPeer chatPeer = (IChatPeer) peers.get(oldName);
  if( chatPeer != null )
  {
    peers.remove(oldName);
    peers.put(newName, chatPeer);
    peer.getPeerConnection().broadcast(new SetNameMessage(oldName, newName));
  }
}

/**
 * Get a list of rooms this server knows of.
 */
public String[] getRooms()
{
  String[] roomList = new String[rooms.size()];
  Iterator i = rooms.keySet().iterator();
  for( int idx = 0; idx < roomList.length && i.hasNext(); idx++ )
  {
    roomList[idx] = (String) i.next();
  }
  return roomList;
}

/**
 * Get a particular room this server knows of.
 * @param name The name of the room.
 */
public IChatRoom getRoom(String name) throws ChatException
{
  IChatRoom room = (IChatRoom) rooms.get(name);
  if( room == null )
```

```
      throw new ChatException( "There is no room named " + name );
    return room;
  }

  /**
   * Create a new room.
   * @param name The name of the room.
   * @param room The room.
   */
  public void createRoom(String name, IChatRoom room) throws ChatException
  {
    if( rooms.get(name) != null )
      throw new ChatException( "A room by that name already exists" );
    rooms.put(name, room);
    peer.getPeerConnection().broadcast(new CreateRoomMessage(name) );
  }

}
```

ChatRoom

A chatroom has a name and a list of members. Chatroom instances are located on the peer where the chatroom was created.

```
/**
 * A chatroom is a container that holds a list of members and manages the list.
 * @author twheeler
 */
public class ChatRoom implements IChatRoom
{
  private String name;
  private List members;

  /**
   * Create a ChatRoom.
   * @param name The name of the room.
   */
  public ChatRoom(String name)
  {
    this.name = name;
    members = new ArrayList();
  }

  public String getName()
  {
    return name;
  }

  /**
   * Join a chatroom.  We have no need to invoke directly on members, so we just
   * take the member name and add it to the list of people in the chatroom.
   */
  public void join(String name) throws ChatException
  {
    if( members.contains( name ) )
      throw new ChatException( name + " is already a member of " + name );
    members.add( name );
  }

  /**
   * Leave a chatroom.
   */
  public void leave(String name)
  {
    members.remove(name);
```

```
  }

  /**
   * Get a list of all members in the chatroom.
   */
  public String[] getMembers()
  {
    String[] membersList = new String[members.size()];
    Iterator i = members.iterator();
    for( int idx = 0; idx < membersList.length && i.hasNext(); idx++ )
    {
      membersList[idx] = (String) i.next();
    }
    return membersList;
  }
}
```

ChatCommand

The ChatCommand interface is implemented by all commands. Commands are entered on the commandline. The name (first word) of the command is used as a lookup to obtain a command-specific factory. The command input is passed to the factory, which creates a ChatCommand instance to execute the command. This design allows new commands to be added easily.

```
/**
 * A ChatCommand is an application-specific command.
 */
public interface ChatCommand
{
  /**
   * Execute the command.  A reference to the application is provided.
   * @param peer The local ChatPeer.
   */
  void execute(ChatPeer peer) throws ChatException;
}
```

ChatCommandFactory

The ChatCommandFactory is a factory for creating commands. Commands entered by the user are given to the appropriate factory (based on a lookup), which then takes the input and creates a command of the appropriate type to be executed.

```
/**
 * A factory for creating commands.
 */
public interface ChatCommandFactory
{
  /**
   * Create a command from the given input.
   * @param peer A reference to the local peer.
   * @param input The command entered.
   * @return The ChatCommand to execute.
   */
  ChatCommand create(ChatPeer peer, String input);
}
```

ChatCommandInfo

This class encapsulates information about a command.

```
/**
 * Encapsulate information about a command: the name, help (brief and verbose), and the
command factory.
 */
public class ChatCommandInfo
{
  private String name;
  private String helpBrief;
  private String helpVerbose;
  private ChatCommandFactory factory;

  /**
   * Create ChatCommandInfo for a command.
   * @param name The name of the command.
   * @param factory The factory for creating the command.
   * @param helpBrief Brief help.
   * @param helpVerbose Verbose help.
   */
  public ChatCommandInfo(String name, ChatCommandFactory factory, String helpBrief, String
helpVerbose)
  {
    this.name = name;
    this.helpBrief = helpBrief;
    this.helpVerbose = helpVerbose;
    this.factory = factory;
  }

  /**
   * Get the name of the command.
   * @return The name.
   */
  public String getName()
  {
    return name;
  }

  /**
   * Get the brief help for the command.
   * @return Brief help.
   */
  public String getHelpBrief()
  {
    return helpBrief;
  }

  /**
   * Get verbose help for the command.
   * @return Verbose help.
   */
  public String getHelpVerbose()
  {
    return helpVerbose;
  }

  /**
   * Get the command factory for the command.
   * @return The factory.
   */
  public ChatCommandFactory getFactory()
  {
    return factory;
  }
}
```

CommandFactory

The CommandFactory is used by the ChatPeer for creating ChatCommands from user input. It also serves as a registry of commands.

```
/**
 * CommandFactory for creating commands from an input string.  Also manages the list of
known commands.
 * @author twheeler
 */
public class CommandFactory
{
  // static fields and methods
  private static List commandList = new ArrayList();
  private static Map commandTable = new HashMap();

  static {
    register( new ChatCommandInfo("exit", new DisconnectPeerFactory(), "Stop the peer.",
"Stop the peer.  The peer is disconnected from the peer network.") );
    register( new ChatCommandInfo("help", new HelpFactory(), "Get help on command(s).", "Get
help on commands.  For extended help specify the command name.") );
    register( new ChatCommandInfo("name", new NameFactory(), "View or change your name.",
"View your name, or change it to the specified name." ) );
    register( new ChatCommandInfo("login", new LoginFactory(), "Login.", "Login to the
server with your chat nickname." ) );
    register( new ChatCommandInfo("send", new SendMessageFactory(), "Send a message.", "Send
a message to someone else." ) );
    register( new ChatCommandInfo("rooms", new ListRoomsFactory(), "List chatrooms", "List
all known chatrooms." ) );
    register( new ChatCommandInfo("create", new CreateRoomFactory(), "Create a room",
"Creates a new chatroom." ) );
    register( new ChatCommandInfo("join", new JoinRoomFactory(), "Join a room", "Join a
chatroom." ) );
    register( new ChatCommandInfo("leave", new LeaveRoomFactory(), "Leave a room", "Leave a
chatroom." ) );
    register( new ChatCommandInfo("members", new RoomMembersFactory(), "List room members",
"Show all users who have joined a room." ) );
    register( new ChatCommandInfo("rsend", new RoomSendMessageFactory(), "Send a message to
a room", "Chat within a chatroom." ) );
  }

  /**
   * Register a command.
   * @param info Information about the command.
   */
  public static void register( ChatCommandInfo info )
  {
    commandList.add(info);
    commandTable.put(info.getName(), info);
  }

  /**
   * Get a list of registered commands.
   * @return An iterator for the list of commands.
   */
  public static Iterator getCommands()
  {
    return commandList.iterator();
  }

  /**
   * Look up a registered command from its name.
   * @param name The name of the command.
   * @return The information about the command.
   */
  public static ChatCommandInfo lookupCommand(String name)
  {
```

```
    return (ChatCommandInfo) commandTable.get(name);
  }

  // fields
  private ChatPeer peer;

  // constructors

  /**
   * Create a new command factory.
   */
  public CommandFactory(ChatPeer peer)
  {
    this.peer = peer;
  }

  // methods

  /**
   * Create a new command from the given input.
   */
  public ChatCommand create(String input) throws ChatException
  {
    String commandName = getCommandName(input);
    ChatCommandFactory factory = null;
    ChatCommandInfo commandInfo = lookupCommand(commandName);

    if( commandInfo != null )
    {
      factory = commandInfo.getFactory();
      return factory.create(peer, input);
    }

    throw new ChatException( "I don't know the command '" + commandName + "'" );
  }

  /**
   * Get the name of the command from the input string.  This is the first word of the input
string.
   * @param input The command input.
   * @return The command name.
   */
  private String getCommandName(String input)
  {
    StringTokenizer st = new StringTokenizer(input);
    return st.nextToken().trim();
  }
}
```

ChatMessage

As mentioned earlier, there are two ways to provide application-specific functionality for peer messages. Our chat application does this by extending the Message class. ChatMessage is the base class for all chat application message subclasses. It declares an invoke() method which is called by the chat application's MessageHandler. Subclasses of ChatMessage implement invoke() to provide functionality specific to that type of message.

```
/**
 * A ChatMessage provides application-specific behavior for messages.
 * See the invoke() method for details.
 */
public abstract class ChatMessage extends Message
{
  /**
   * Create a ChatMessage with the given arguments
   * @param arguments The arguments.
```

```
   */
  public ChatMessage( Object[] arguments )
  {
    super( "com.recursionsw.chat.message", arguments );
  }

  /**
   * Invoke the command.  Implemented by subclasses to provide application-specific
functionality.
   * @param peer A reference to the ChatPeer application.
   */
  public abstract void invoke(ChatPeer peer);
}
```

ChatMessageHandler

The ChatMessageHandler handles all chat messages. It is registered with the peer's MessageDispatcher on application startup, and dispatches all chat messages by calling their invoke() method.

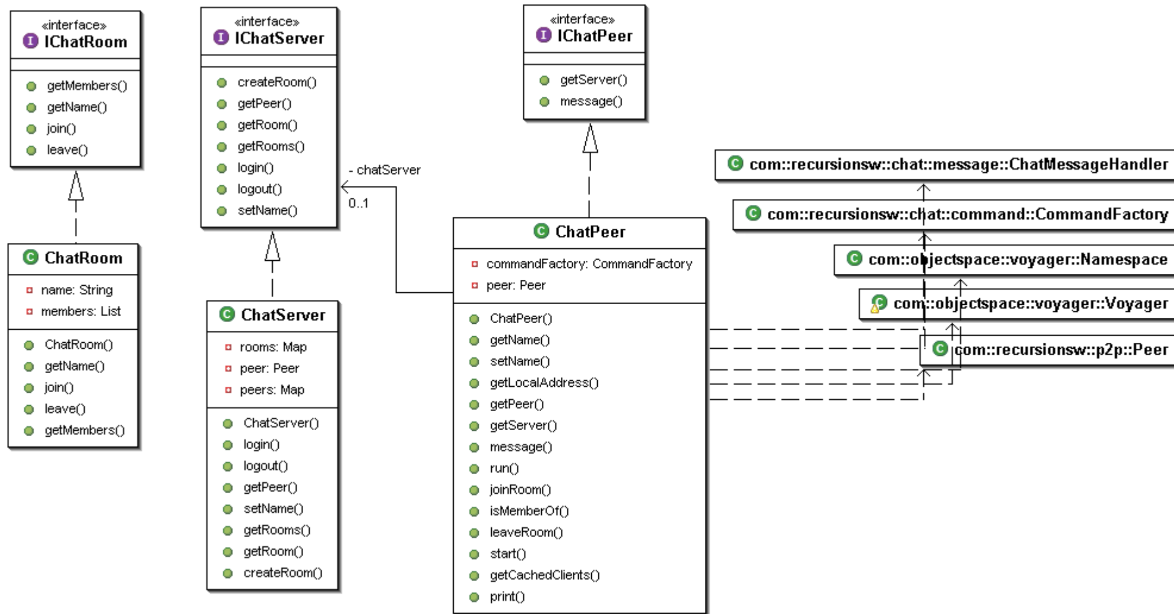
```
/**
 * The ChatMessageHandler is registered with the MessageDispatcher and handles all
 * messages for the chat application.  Chat messages extend ChatMessage, which
 * declares invoke().
 */
public class ChatMessageHandler implements MessageHandler
{
  private ChatPeer chatPeer;

  /**
   * Create a ChatMessageHandler.
   * @param chatPeer A reference to the chat application.
   */
  public ChatMessageHandler( ChatPeer chatPeer )
  {
    this.chatPeer = chatPeer;
  }

  /**
   * Invoke the chat message.
   * @param message The message to handle.
   */
  public void handleMessage(Message message) throws PeerException
  {
    ChatMessage chatMessage = (ChatMessage) message;
    chatMessage.invoke( chatPeer );
  }
}
```

Following are class and sequence diagrams showing major classes in the chat application and their dependencies.

Chat class diagram – com.recursionsw.chat

This package contains the core application classes.

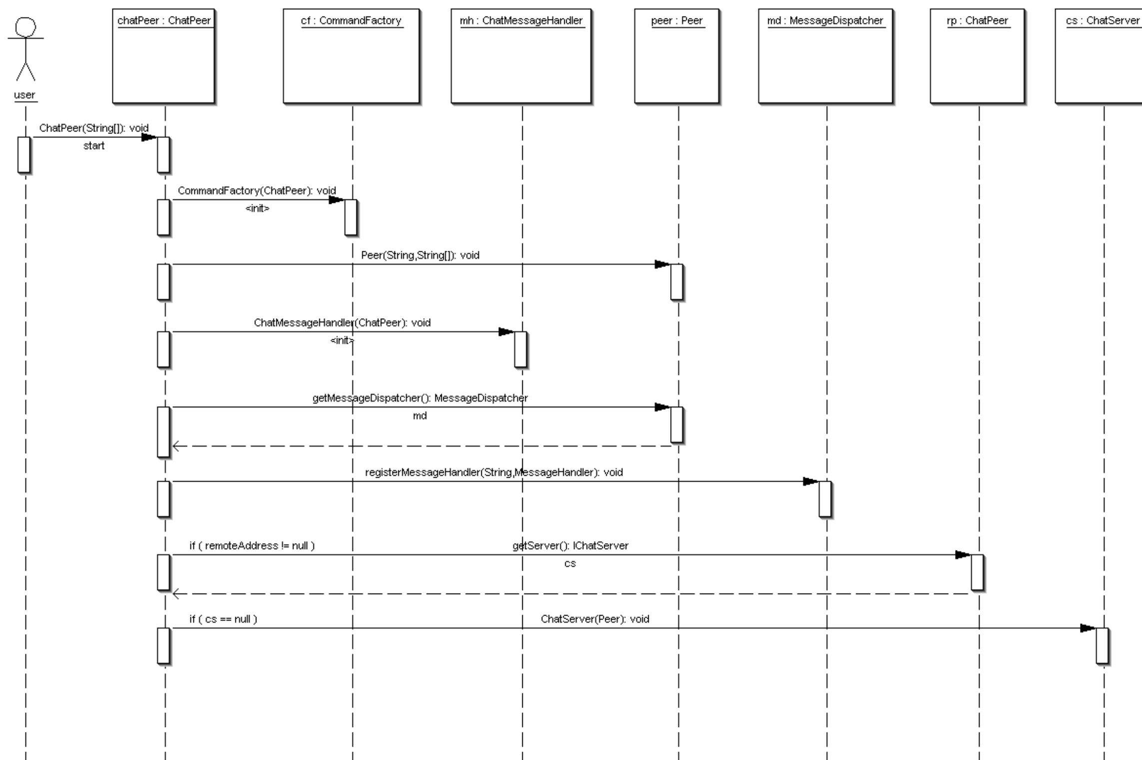
Figure 3. Core Application Classes



Chat class sequence diagram

This diagram shows what happens on application startup.

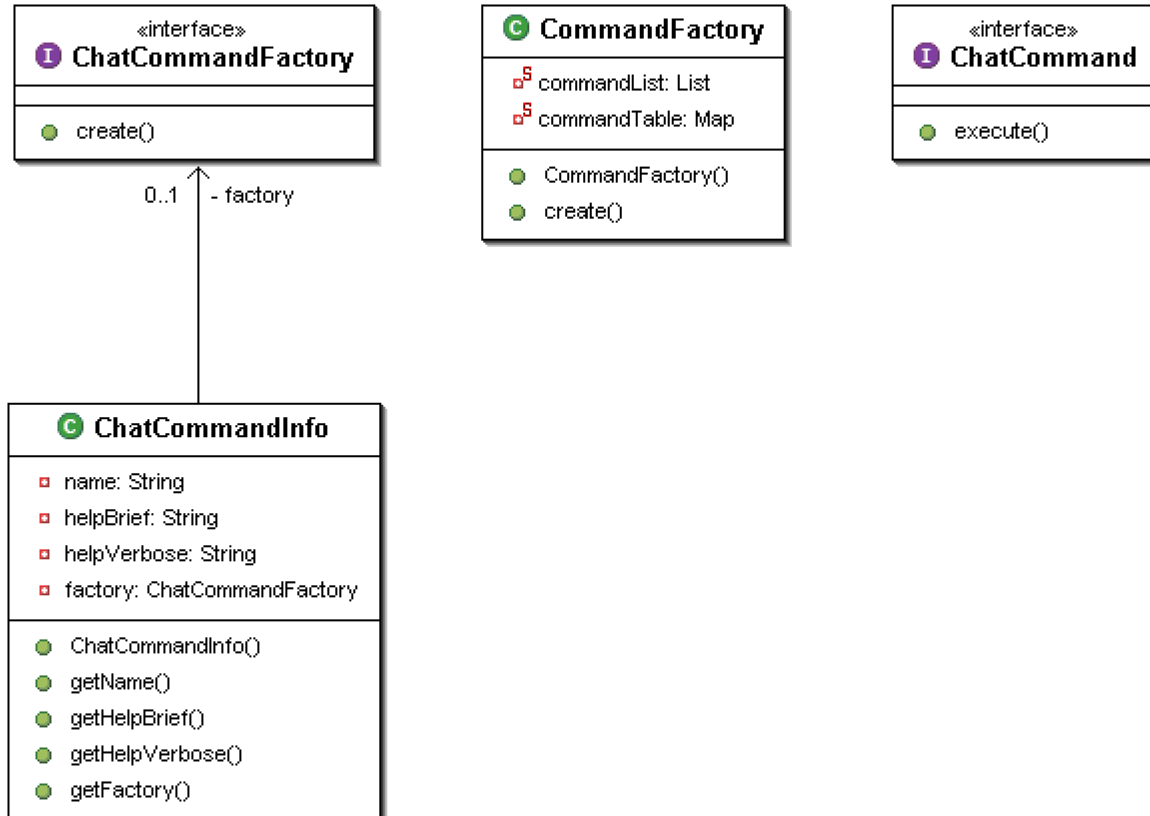
Figure 4. Core Application Classes



Chat command class diagram – com.recursionsw.chat.command

The chat application uses the Command pattern to handle commands entered as input. This diagram shows the primary interfaces and classes for the management and dispatch of commands.

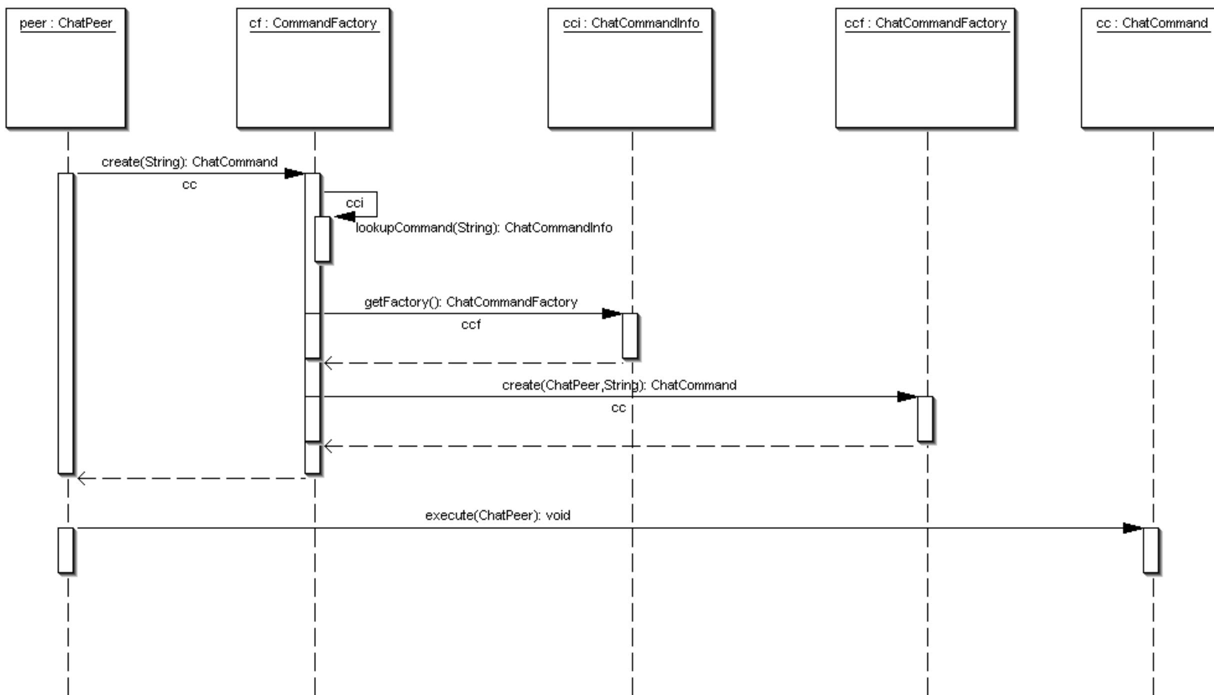
Figure 5. Primary Interfaces and Classes



Chat command sequence diagram

This diagram shows the sequence of events when the user enters a command at the console.

Figure 6. Command Sequence of Events



Running the Chat Application

To demonstrate how everything works together, let's start the application. We will have three users, George, Fred, and Robert. Open three command prompts and configure the classpath for each to include both Voyager and the voyager_chat.jar file. George starts up first:

```
>java com.recursionsw.chat.ChatPeer
Peer started on //TWHEELER-2K:10102
?:login George
George:
George has joined
George:
```

When George starts up, he knows he is the only person in the chat group. As a result, his node starts as both a peer (ChatPeer) and a server (ChatServer). The default port for a peer is 10102. Consequently, he will login on his own node. Next, Fred joins:

```
>java com.recursionsw.chat.ChatPeer //localhost:10102
Peer started on //TWHEELER-2K:1539
?:login Fred
Fred:
Fred has joined
Fred:
```

Notice that a message has been printed on George's console notifying him that Fred has signed on. How did this happen?

When Fred started his peer, he specified a (one-element) list of peer candidates which his peer attempted to connect to. The ChatPeer will use Voyager's Naming Service to attempt to look up a remote peer at each of the candidate addresses. On the first successful lookup, it asks the remote peer for its ChatServer. In this case the remote peer node is also the ChatServer node. Fred logs in to this node. On a successful login, the ChatServer broadcasts a message to all peers notifying them that a new user has joined the group. Voyager's Space feature is used to perform the broadcast in a scalable manner.

Next, Robert joins:

```
>java com.recursionsw.chat.ChatPeer //localhost:1539
Peer started on //TWHEELER-2K:1556
?:login Robert
Robert:
Robert has joined
Robert:
```

Here we demonstrate the location transparency of the server. Robert includes in his candidate list the address of Fred's peer, //localhost:1539. Robert is able to successfully log in even without knowing the address of the server.

Now that all three peers have been started, it is easy to message between them:

```
Robert:send Fred Good morning!!
```

Robert's message is displayed on Fred's console:

```
Robert: Good morning!!
```

This message is sent directly from Robert's peer to Fred's peer. Robert's peer gets a reference to Fred's peer from the ChatServer. All further communication from Robert to Fred is direct: it does not go through the server.

Next, George and Fred join the Global chatroom. (This chatroom is created automatically by the ChatServer on startup.)

```
George:join Global
George has joined Global
George:
Fred has joined Global
George:
```

```
Fred:join Global
Fred has joined Global
Fred:
```

Notice that when George joins Global, no one else receives notification; when Fred joins, George gets notified that Fred has joined Global. This is because only members of the chatroom receive notifications of events that happen to that room.

Now, Fred and George can communicate in the chatroom:

```
Fred:rsend Global What are we doing for lunch?
Global : Fred: How do you like Robert's new car?
Fred:
Global : George: I vote pizza!
Fred:
```

```
George:
Global : Fred: How do you like Robert's new car?
George:rsend Global I vote pizza!
Global : George: I vote pizza!
George:
```

Unlike the messages sent directly from Robert to Fred, messages sent to a chatroom are propagated to all peers. Anyone in the chatroom will receive the message and print it to the console.

Spend a few minutes with a few co-workers experimenting with the application. As a tutorial, it does not have the robustness of a production application, but you should be able to see how location transparency, message broadcasting, and direct peer-to-peer connectivity work together.

Conclusion

Summary

P2P applications require a core set of infrastructure services, including resource publication; resource location; and resource utilization. Voyager's peer-oriented architecture and advanced features, including Universal Naming Service and Subspace, make it possible to rapidly develop the infrastructure services required by a P2P application. The chat application we have developed demonstrates how Voyager can be used to rapidly and easily create P2P applications.

Many other potential P2P applications can be built with Voyager:

- *Distributed Computing:* A P2P distributed computing application can be rapidly developed by leveraging Voyager's ability to move objects and dynamically load classes across the network.
- *Collaboration:* By broadcasting changes in application state and user input to remote peers, each peer can present a consistent visual presentation to the user. Distributed editing, whiteboarding, and training are three possible uses.
- *Resource and Data Replication:* A flexible, fault-tolerant distributed cache can easily be developed using Voyager's Space feature. Data or files traditionally hosted on a fileserver can be spread to multiple peers, utilizing unused disk space and reducing network congestion to the fileserver.

Voyager is a flexible, powerful framework for the development of distributed applications. No other distributed computing framework is as easy to use or provides the same set of advanced features.

About the author

Thomas Wheeler is a senior software engineer at Recursion Software, Inc. He may be contacted by email at engineer@recursionsw.com.

Copyright © 2003 Recursion Software, Inc. All rights reserved. Voyager is a registered trademark of Recursion Software, Inc. All other trademarks or service marks are the property of their respective holders.

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com

RECURSION SOFTWARE, Inc.