OogP2P Framework Documentation

Sanjay Ginde[1]
Duke University

David Goldberg[2]
Duke University

Chris Zeiders[3]
Duke University

December 4, 2003

[1]sjg@duke.edu
[2]dtg@duke.edu
[3]cmz@duke.edu

Abstract

This paper serves as documentation for the OogP2P framework, a framework for developing and running applications on a peer-to-peer system. The OogP2P framework is designed in such a way that users can develop and run new applications while having to add a constant, O(1), amount of code to the existing framework, without any modification to pre-existing applications running on the system, and without any modification to the network model. "Applications" are not limited to programs such as file sharing, chat, and networked games, but also include peer-to-peer components such as algorithms for searching, load balancing, and network-data gathering, to name a few. Developers can create applications by following the Communication and Application Architectures as specified in this paper.

The OogP2P framework provides the base networking protocols for a decentralized peer-to-peer network and any applications developed from our framework are geared to run on this network.

Contents

1 Overview **2**

2 Network Model **2**

2.1 Server 2

2.2 Client 3

2.3 Connecting 3

2.4 Network Communication 3

3 Communication Architecture **3**

3.1 Overview 3

3.2 Client Administrator 4

3.3 Nettable 5

3.4 Info Received 5

3.5 Nettable - InfoReceived Relationship 5

3.6 Simple Communication 6

3.7 Complex Communication 7

4 Application Architecture **7**

4.1 Overview 9

4.2 Application Encapsulation 9

4.2.1 Nettables 9

4.2.2 InfoReceived Objects 11

4.2.3 Factory for InfoReceived Objects 11

4.2.4 Commands 11

4.3 Application Interface and Layer 12

4.4 Utilities Map 14

4.4.1 Administrators 14

5 How to Write a New Application: Chat Program **15**

5.1 Chat Window 16

5.2 Chat Administrator 18

5.3 Chat Application 19

5.4 Chat Factory 19

5.5 Communication Architecture for Chat 20

5.5.1 Initializing Connections 21

5.5.2 Sending Messages to Connections 24

5.5.3 Closing Connections 27

5.6 Chat Class and Package Descriptions 27

6 Future Work **28**

1 Overview

The OogP2P Framework provides two fundamental architectures:

1. Communication Architecture
2. Application Architecture

The Communication Architecture provides for a simple way for data to be sent between peers on the network. All data sent between peers is encapsulated inside a Nettable object. Upon receiving a Nettable object, a peer will construct a corresponding InfoReceived (IR) object that will perform the actions requested by the Nettable object. The Communication Architecture is discussed further in section 3.

The Application Architecture provides a means to develop new applications to run on the OogP2P network while maintaining these three main characteristics:

1. No modification to the existing network model
2. No modification to any other applications already running on the network.
3. Strictly a constant, O(1), addition of code (i.e., no modification of existing code) to the framework when a new Application is added.

All the new classes that are created to implement a certain application are encapsulated inside an Application object. These classes that are 'housed' inside the Application object include all the classes to implement the domain of the new application, as well as the Nettables and InfoReceived objects created to ensure compatibility with the Communication Architecture. There is complete separation between the various applications running on the network-all work independently of each other. The Application Architecture is discussed further in section 4.

2 Network Model

In truly decentralized peer-to-peer networks, there is no central server-every node on the network acts as both a server and a client. Within this paradigm we designed a simple networking model with the idea of each computer acting both a server and a client.

2.1 Server

The Server is integral to the peer-to-peer system as it allows nodes to connect and request information. When a node connects to the Server, a Connection object is created which stores basic information of the connecting node. Once the connection is established, the connected node, now a client of the server, can request information from the server node.

2.2 Client

When a node is requesting information it creates a Client object to connect to the node where the information is. Once this client connection is established, the client node can then request information from the node to which it just connected.

2.3 Connecting

Under the current network architecture, when a new node wants to join the peer-to-peer system, it attempts to make a TCP/IP connection with an IP address of a node that is already on the network. This introduces the problem of peer discovery. In the current system, each computer possesses a file of IP addresses of nodes on the system. Under a controlled environment, the maintenance of this file is simple, but in a widespread or "real world" running of the system, this maintenance can be very difficult. This peer discovery problem is a current area of research in peer-to-peer systems, and was not the focus of this project.

2.4 Network Communication

The basic structure of network communication is very simple. Communication between nodes occurs through ObjectOutputStreams and ObjectInputStreams. Generally speaking, the Server handles incoming data Client deals with outgoing requests. Under this network model, any type of data can be sent over the network, but as discussed in Section 3.3, the OogP2P Framework restricts this data to be wrapped in a Nettable object.

3 Communication Architecture

The Communication Architecture provides a simple means of sending information across the network in an organized fashion with the goal of maintaining the following characteristics:

1. General interface to the Network Architecture
2. Polymorphic structure for simple filtering of information to specific applications
3. General structure to allow for the development of a wide-range or peer-to-peer components.

3.1 Overview

Since the OogP2P-Framework strives for an easy to use Communication Architecture, it provides a ClientAdministrator object to interface with the Network Architecture. (See Figure 1) The ClientAdministrator acts as a Facade [1] between the Oogp2p framework and the underlying network model. That allows

Figure 1: Client Administrator

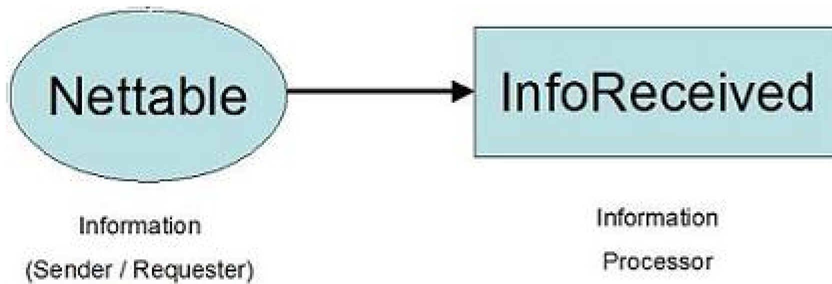

Figure 2: Nettable-InfoReceived Relationship

applications to not have to deal with creating the actual network connections with new peers or worrying about how to send the data to them.

Communication between nodes on the peer-to-peer network is facilitated by two objects, the Nettable and InfoReceived (IR). (See Figure 2) At the most abstract level, Nettables are the information packets that get sent over the network. They either request or send information from one node to another. When one is received, a corresponding, parallel IR object is created to process and perform the action requested by the associated Nettable. For example, a ChatInitNettable is processed via a ChatInitInfoReceived object.

3.2 Client Administrator

The ClientAdministrator was created to allow an easy to use interface with the Network Architecture. The ClientAdministrator allows for the swapping in and out of different Network Architectures as it provides a general interface to it. Also, it encapsulates all the network client connections in one place so all the applications can use them, but not manipulate them, through collection of methods.

The most important method of the ClientAdministrator class is the method:

```
boolean sendNettable(IP address, Nettable object)
```

Whenever an application wants to send a Nettable over the network, all it needs

to do is simply call this function. The ClientAdministrator takes care of creating the connection if necessary and actually sending the information off, returning whether it succeeded or not.

3.3 Nettable

Essentially, Nettables can be viewed as wrappers, or as an Adapter [1] around information so that the data can fit easily within the framework. All information over the network must extend from the Nettable object. This provides some advantages. For example, inherent in all Nettables is the sender and receiver information so receivers of information can know exactly where it came from. Also, the basic Nettable provides a getContents() method, so a client receiving the Nettable can extrapolate the pertinent information out easily.

Another reason for having all networked information inheriting from the Nettable object is that it allows for the framework's applications to work with a common interface when generating the appropriate IR object. (More on the Nettable hierarchy and its relation to the Application Architecture can be found in section 4.2.1.)

3.4 Info Received

As stated in the Communication Architecture overview above, Nettables and InfoReceived objects work in a parallel fashion. For every Nettable object that either requests or contains information, an InfoReceived object is instantiated to process it. Because of the parallelism, the InfoReceived objects also have a symmetrical hierarchy structure to the Nettables.

Each InfoReceived objects perform a specific task, which is implemented in the execute() method. Typically this task will involve some manipulation of the other objects of the specific application, such as mapping incoming text to the appropriate chat window or updating node information in a Chord application. The InfoReceived objects are essential to the application structure, for an InfoReceived object is created whenever a client receives a Nettable. (For more information on the IR relationships with the Application Architecture see section 4.2.2).

3.5 Nettable - InfoReceived Relationship

At first glance, it may seem redundant and unnecessary to have the parallel constructions of Nettables and InfoReceived objects, thinking that they could easily be combined into one object. The need for separation however is clear - when data is sent across the network, it is asking the receiver of the data to perform some sort of action, and to perform that action the receiver will need to manipulate and use its *local applications*.

For example, user A could request a chat with user B by sending user B a ChatInitNettable over the network. User B would then instantiate a ChatInit-InfoReceived object locally to process that request. The local ChatInitInfoRe-

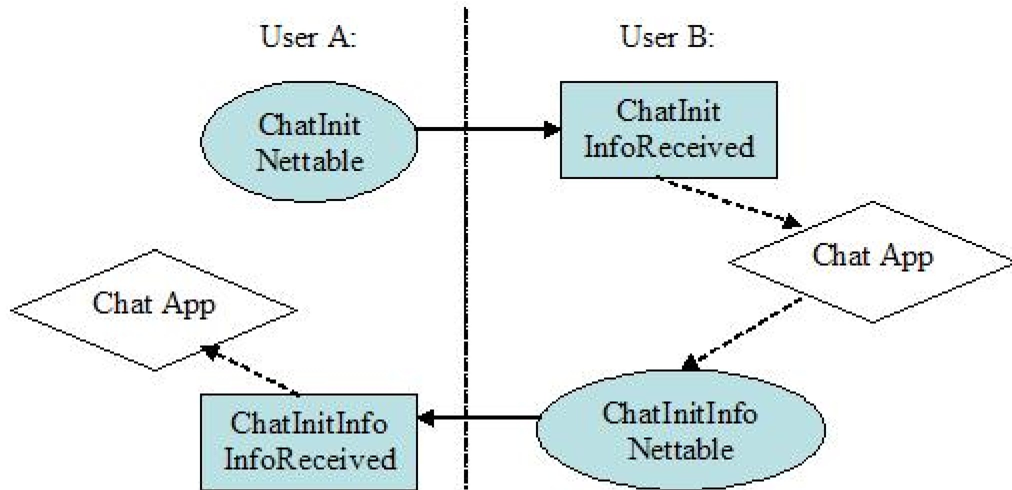

Figure 3: Simple Communication

ceived object would then manipulate user B's local chat application and decide whether it wants to chat with user A create the appropriate window, etc. Furthermore, from a security and good-design perspective, user A cannot directly manipulate user B's applications and objects - user A should merely request user B to manipulate its objects, thereby giving user B the option of accepting or rejecting the request.

3.6 Simple Communication

Simple information requests are easily facilitated through the Communication Architecture. Figures 3 and 4 show an example of a simple communication scheme involving chat initiation request from user A to B. User A sends out a ChatInitNettable to user B, containing information such as user name to allow user B to form a proper chat connection. User B receives the ChatInitNettable request and instantiates the appropriate parallel InfoReceived object, a ChatInitInfoReceived. This InfoReceieved object can then process the request using user B's local chat application. If user B wanted to chat with user A, then he or she would simply sent back a ChatInitInfoNettable (essentially an affirmative response) which would house the necessary information to allow user A to form a chat connection with B. User A would then instantiate the appropriate parallel InfoReceived that would utilize the local user A chat application and complete the process.

Nettable	Nettable's Data	Parallel IR	IR's Execution
Chat Init Nettable	User name, IP, port, etc.	ChatInitIR	1. Chat initialization 2. Responds with ChatInitInfo Nettable
Chat Init Info Nettable	'Yes' response, user name, IP, port, etc.	Chat Init Info IR	1. Chat initialization

Figure 4: Simple Communication: Nettable-InfoReceived Relationship

3.7 Complex Communication

As mentioned earlier, the Nettable-InfoReceieved architecture was envisioned to be abstract enough to allow complex network communication between peer nodes, such as when the target for a given request is not known. This type of communication is crucial, as on P2P networks, information is never stored in a central location to facilitate a simple request. Often, requests must be passed on and routed between nodes indeterminately until the target is reached. For example, in a simple file search, the location of the file is not known. The file request is simple passed on between nodes until a target node containing the requested file is reached.

Figures 5 and 6 show an example of a file search on the Chord P2P network structure. Node 1 is requesting a file (legally, of course) through a ChordFindFile Nettable. The request is sent to node 2, which in turn processes the file request with the parallel ChordFindFile InfoReceived object. If node 2 has or knows where the file is, then it simply sends back a ChordFileFound Nettable; nothing special here. However, chances are that node 2 does not have or know exactly where the file is. It then passes the request on to a more appropriate node, say node 3 (via the Chord algorithm) that has a better chance of knowing where the file is. Node 3 then acts in similar fashion to node 2: checking to see if it knows where the file is or passing it on indeterminably to some node x that knows where the file is. Once that node x is reached, it can then send back the ChordFileFoundNettable back to the originator of the request, node 1.

4 Application Architecture

The Application Architecture provides a means to develop new features to run on the OogP2P network while maintaining these three main properties:

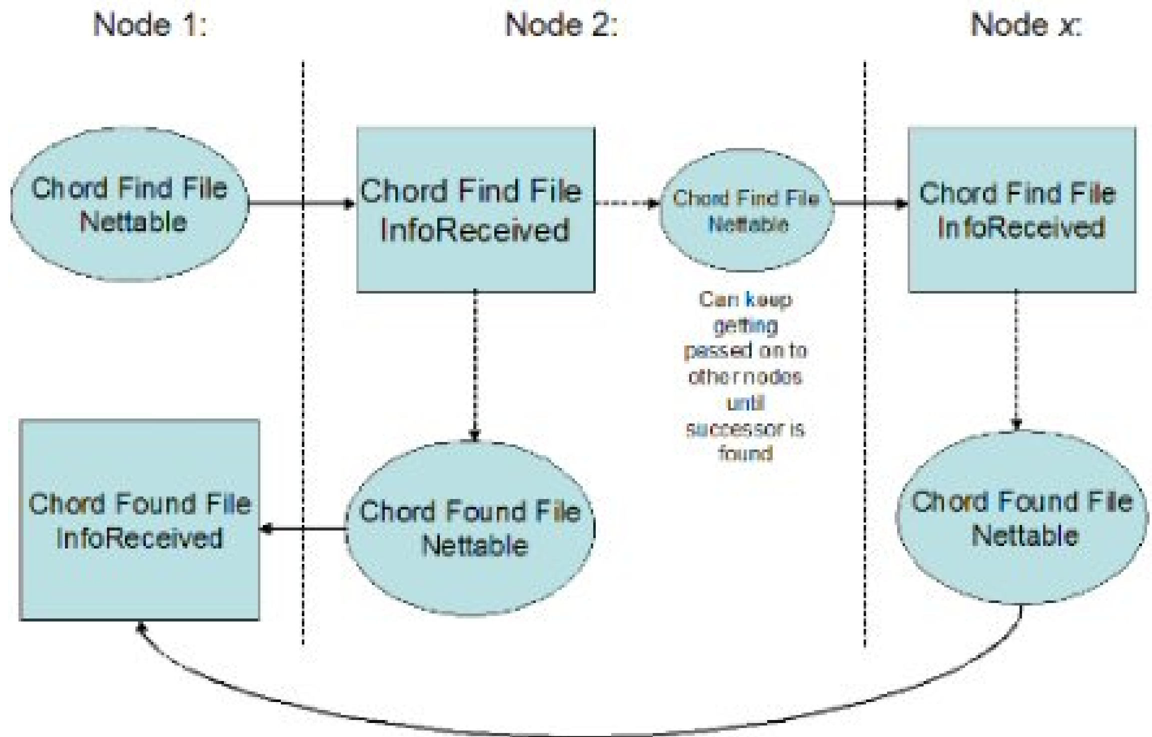

Figure 5: Complex Communication

Nettable	Nettable's Data	Parallel IR	IR's Execution
Chord Find File Nettable	File name, request originator, etc.	Chord Find File IR	1.If location of file known, responds with Chord File Found Nettable, or… 2.Passes the find file request on to other nodes
Chord Found File Nettable	Location of where file is at	Chord Found File IR	1.Displays where the file is located

Figure 6: Complex Communication: Nettable-InfoReceived Relationship

1. No modification to the existing network model
2. No modification to any other applications already running on the network.
3. Strictly a constant, O(1), addition of code (i.e., no modification of existing code) to the framework when a new feature is added.

4.1 Overview

Most developers typically view features as user-ended programs, such as tic-tac-toe, chat, file sharing, word processing, etc. But the OogP2P Framework supports much more than just these user-ended programs; it also supports components of p2p networks-such as searching algorithms, load balancing algorithms, means for statistical analysis of the network, and network structuring to name a few. The OogP2P framework treats these non-user-ended applications in the exact same way as user-ended applications.

4.2 Application Encapsulation

In order to accomplish the three goals stated above, we developed a general Application object from which all specific types of Applications will inherit. This Application object encapsulates all areas of an application-from the Nettables and InfoReceived objects needed to maintain the Communication Architecture of the OogP2P Framework to the classes specific to the domain of the particular application. (See Figure 7.)

4.2.1 Nettables

The design of having each Application house the Nettables associated with that particular application forces each Nettable to be associated with exactly one application. In other words, one Nettable cannot be handled by more than one application. This aspect of the framework design provides for a firm distinction between the multiple applications running on the system and allows for two important properties of the system:

1. The addition of new applications will not be able to inadvertently intercept Nettables meant for pre-existing applications, thereby avoiding introducing bugs in applications that already work.
2. Safeguard against applications executing Nettables that they do not know how to handle.

A corollary to property 1 is that multiple applications that span similar domains can run at the same time. For example, two DHT searching algorithms, Chord [3] and Pastry [4], can both run on the system at the same time. Since the code for Chord and Pastry are encapsulated inside separate and distinct Applications, there is no possibility for a "mix-up" of search-data being processed on the client. One can then develop unique and powerful searching-optimization

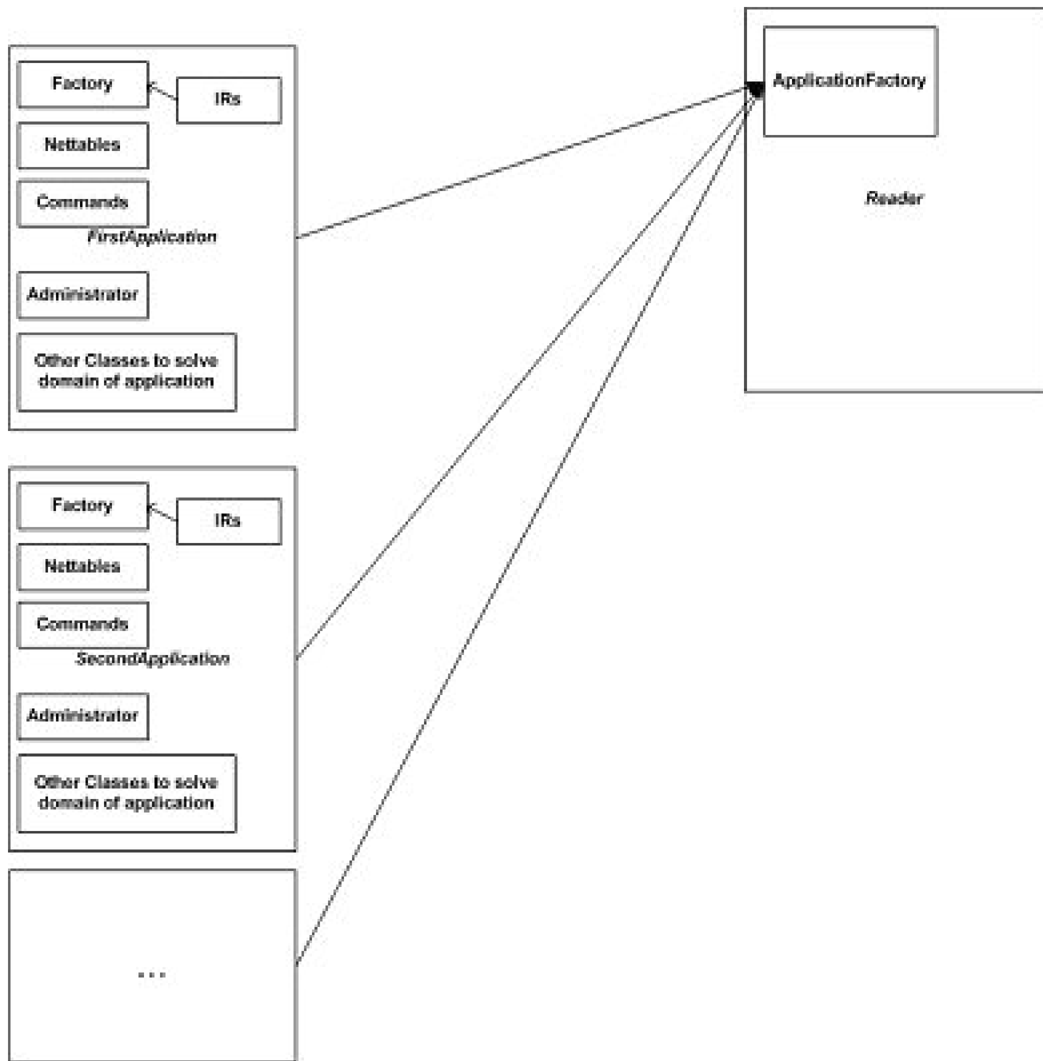

Figure 7: Application Encapsulation

algorithms that can possibly combine the strengths of the multiple searching applications running on the same client.

4.2.2 InfoReceived Objects

In addition to the classes and code needed to satisfy the domain of the application, InfoReceived objects are necessary for any cross-network communication. In many applications, the InfoReceived objects drive the application-they utilize the classes that were created to solve the domain of the problem. The execute() function of every InfoReceived performs a set of operations to facilitate the execution of the application. Below is an example from the Chord Application. It is the execute() code from the AddKeyToTableIR, which given a the key of a file, inserts that key into its table of keys.

```
public void execute() {
    ChordAdministrator chordAdmin =
        ChordAdministrator)p2p.getUtils().get(p2p.CHORDADMIN);

    //insert the key into the map which maps the key
    //to the originator's IP
    chordAdmin.addKey(myKeyToInsert, myOriginator);
}
```

NOTE: The myKeyToInsert and myOriginator variables were extracted from the AddKeyToTableNettable. The ChordAdministrator and Utilities map (p2p.getUtils()) is explained below in section 4.4.

4.2.3 Factory for InfoReceived Objects

Just as Nettables are encapsulated within the Application architecture, so are the InfoReceived objects. In order to elegantly handle the addition of new InfoReceived objects into an application and to have it seamlessly work with the existing framework, we have developed a coding practice following the Factory pattern [1]. Using the Factory pattern allows us to isolate the rules of when to use objects from the logic of how to use the objects. This Factory will contain all the IRs associated with the current application. This Factory is instantiated in the specific Application class, whereby the application can determine whether or not it can handle a given Nettable.

Using this design, when a new IR is created for an application at most one line of code needed to be added to the existing code. For example, when the AddKeyToTableIR is created, the following line of code is added to the ChordFactory:

```
myChordIRs.add(new AddKeyToTableIR());
```

4.2.4 Commands

Commands are what the user types into the console window. They primarily initiate an application or play a role in the execution of an application. For

example, the user can type "findUsers" in the console window in order to see all the users currently connected on that network.

Command objects are the same as standard InfoReceived objects but since they are entered by the user via the console window, we determined that it would be a good idea to make them "special in name," even if they are exactly the same in code to regular InfoReceived objects, since in later work we might end up treating commands very differently than other InfoReceived objects.

The processing of commands is a little bit odd in the OogP2P Framework. Under the current implementation, the output of the console window is directly tied to the client to which you are connectedthus the processing of the commands is done on that client. This is counterintuitive since commands should be processed locally and this is an area that is addressed in Future Work (section **??**) below.

4.3 Application Interface and Layer

This Application object works as a layer in between the Communication Architecture and the applications running on the network.

The code for this interface is below:

```
/**
 * This class is the super class for all types of Applications.
 * One application will be created for each application that will
 * be implemented over the framework.
 */
public abstract class Application
{
    abstract public void Init();
    abstract public boolean isType(Nettable n);

    public void process(Nettable n)
    {
        System.out.println("The nettable " + n.getClass().getName()
                           + "cannot be processed.");
        System.out.println("Application <" + n.getApplicationName()
                           + "> not supported");
    }

    abstract public Application makeType(Nettable n);
}
```

When Nettables are received by the Reader thread on the server, the Reader determines which Application that Nettable is associated with, and then sends the Nettable to that specific Application. This Application Interface/Layer provides for some elegant code in this Reader function:

```
    public void run()
    {
        Nettable recvNettable = null;
```

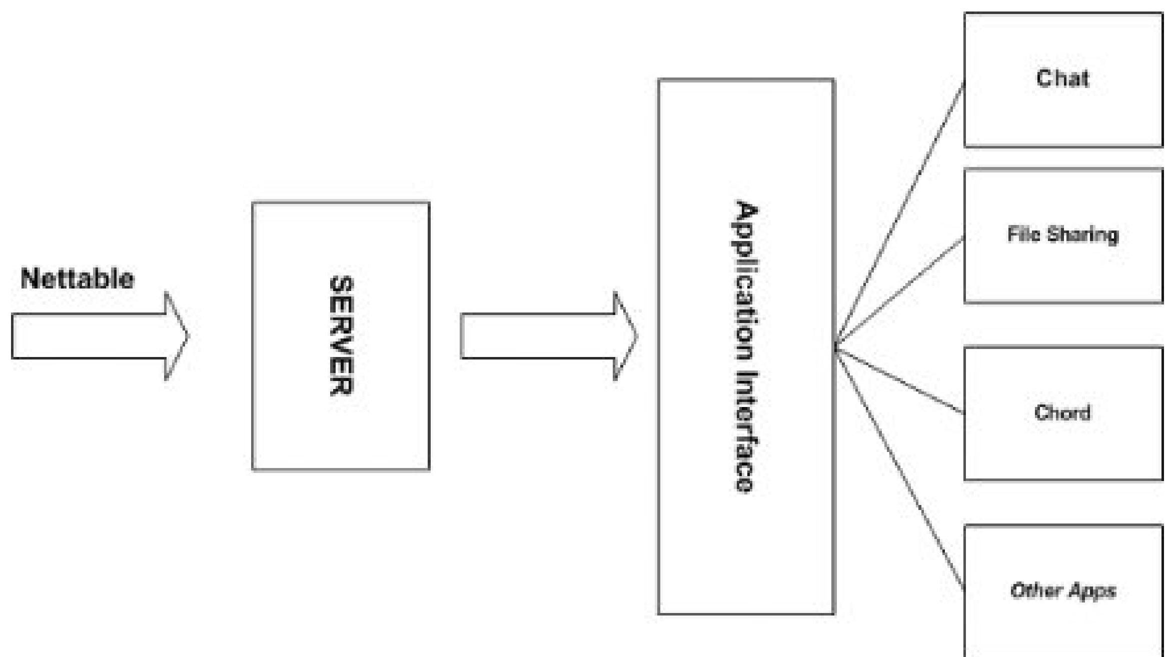

Figure 8: Application Interface

```
    while ( RECEIVE_NETTABLES )
    {
        recvNettable = getNettable();

        Application app =
            myAppFactory.getApplication(recvNettable);
        if ( app != null )
            app.process(recvNettable);
        else
            System.out.println("This Nettable is not supported");
    }
}
```

There is no need for any modification to this networking code when new applications are added to the system. As you can see from line 8, this is accomplished by taking advantage of the Factory Pattern [1]. The Application Factory has an instantiation of every application the client supports, and upon receiving a Nettable, the Application Factory determines which application can handle it. When a match is found, the Factory returns an instantiation of the particular application. The Nettable is then sent to that particular application to be processed. If no Applicaton is found that can handle the Nettable, then a message is reported and the Nettable is thereby discarded.

Furthermore, when a new application is developed and inserted into the system, only one line of code needs to be added to the Application Factory:

```
myApps.add(new myApplication());
```

4.4 Utilities Map

Often times in order for an application to work properly, there is a need for an object to be instantiated once and then used by other classes at various times. One can view such an object in the same light as a "global property" or an environment variable. For example, when an InfoReceived object of the Chord application needs to update its properties so that it can maintain its ring structure, it will need to update the information in a single place so that it will be reflected in other parts of the chord application.

The OogP2P Framework provides a Utilities Map to handle such instances. The Utilities Map is a static map that can be accessed from any part of the system. Currently, this map contains utilities *Administrators* for applications such as Chat, Chord, Pastry and also a Client Administrator to facilitate client/server/networking duties.

4.4.1 Administrators

The *Administrator* object (Figure 9) is a convention that we developed that is not explicitly necessary to develop applications to run on the OogP2P network, but serve as a good OO-design that makes implementing new applications easier.

Figure 9: Administrator

An Administrator serves as a mediator between the InfoReceived objects and the objects that were created for the scope of the application. [1]

Instead of the InfoReceived object directly manipulating the objects of the application, it will go through the Mediator. The primary benefit of this design is that for each application, typically only the application's Administrator needs to be added to the Utility map. The Mediator promotes loose coupling by keeping the domain objects and the InfoReceived objects from referring to each other explicitly, and it lets you vary their interaction independently.

This mediator is stored in the Utilites Map in the main p2p.java class, whereby all of the InfoReceived objects can have access to it.

For example, in the Chat Application, the ChatMessageIR needs to display the received text onto the chat window of the user. It does this through the ChatAdministrator which serves as a mediator between the IR and the chat sessions the application is currently supporting.

```
public void execute()
   {
       ChatAdministrator chatAdmin =
           (ChatAdministrator)p2p.getUtils().get(p2p.CHATADMIN);
       ChatWindow cw = chatAdmin.getSession(myFrom, myFromIP);

       cw.addText(myFrom, myMessage);
   }
```

5 How to Write a New Application: Chat Program

Writing a new application is relatively simple and can be readily examined in a cookbook fashion. The first step, of course, will be to decide which application to write. For this example, we will look at a chatting application in which people logged on to the p2p network can chat with others who are logged on.

- The first step is creating the package. In this case it might be called oogp2p.chat.

This package will house all the necessary classes for the chat application. These classes need to be divided into three separate categories:

- Those classes relating to the communication architecture that are used specifically to communicate data from one Client to another
- Those classes relating to the application architecture which are used to create and distribute data within each application and then if needed, send that data to the correct client in the right package.
- Those classes relating to the domain, that is, classes needed to implement the application.

Now that we have determined the basic distinctions between what we need we will need to examine each of those in some detail. First, we must look at the Communication Architecture (Section 3). Each feature will need its own set of Nettables, Commands and InfoReceived objects as each will serve a different purpose. Thus, for each application, we must determine the important components of its functionality and build those into the Communication Architecture. At a simplistic level, it seems our Chat application will need to be able to do the following things:

- Initiate a Connection with one (or more) people
- Correctly send messages over any of those Connections
- Close a Connection, or close all connections

So how would one implement each of these functionalities? There are a few components that are basic, but essential in making a viable chat application.

5.1 Chat Window

The first of these is the ChatWindow itself. The ChatWindow will need to be able to take text that is input and pass it to other windows that wish to see what the message was. It must also be able to display text that the client receives as part of a chat message. We accomplish this using a couple functions. Finally, we need a way to determine which chat window belongs to which chat conversation. We first initialize the output streams to point to the clients we want to send messages to like so:

```
private void setOutputStreams(ClientAdministrator clAdmin,
                              InetAddress inetAddr)
{
    myPeerOutputs = new HashMap();
    for (int i = 0; i < myPeerInfo.size(); i++)
    {
        Peer current = (Peer)myPeerInfo.get(i);
        if ( inetAddr.equals(current.getInetAddress()) )
            me = current;
```

```
        else {
            Client c = clAdmin.getClient( current.getInetAddress() );
            myPeerOutputs.put(current, c.getOutputStream());
        }
    }
}
```

We add all the targets to a map and set ourselves so we know not to send messages there. Then to actually send messages, we use this code:

```
// Get text
String message = myInputText.getText().trim() + "\n";

// Add to own window
addText( me.getID() , message );

// Write to each of the outputs
for (int i = 0; i < myPeerInfo.size(); i++) {
    Peer p = (Peer)myPeerInfo.get(i);
    if ( ! p.equals(me) ) {
        ObjectOutputStream out =
            (ObjectOutputStream)myPeerOutputs.get(p);
        try {
            out.writeObject(new MultiChatMessageNettable(me, p.getInetAddress(),
                            myPeerInfo, message));
            out.flush();
        }
        catch (IOException e) {
            System.err.println("Error sending msg in MultiChatWindow");
        }
    }
}

clearEditorWindow();
```

In essence, we merely cycle through our list of peers, checking to make sure that they are not ourselves, and then if not, send a message to that peers through its own output stream.

When we want to add text to the window, we merely call `ChatWindowName.addText("text")` and utilize this function in the ChatWindow class:

```
    public void addText(String userName, String text)
    {
        myChatText.append(userName + " >> " + text);
    }
```

So now we have the basic functionality for the actual window. In order to make each window unique, we include a constructor that includes a from name and from IPaddress, and to name and to IPaddress as well as the stream that goes to the to address. We do that as follows:

```
public ChatWindow(String user, InetAddress userIP,
                String from, InetAddress fromIP,
                ObjectOutputStream outputStream)
```

5.2 Chat Administrator

Now that we have the basic window that allows us to send messages from it, we need to have a way to keep track of how to map messages to the correct chat window. We will need to be able to do the following:

1. Add a new ChatWindow to the vector of ChatWindows that are currently open
2. Create a new session which creates and then adds a new ChatWindow to the vector of currently open ChatWindows
3. Access a specific session or group of sessions
4. End sessions by closing the window.

To accomplish the first goal of being able to add a new chat window, we can use the following code:

```
public void add(ChatWindow cw)
{
    myChatSessions.add(cw); //myChatSessions is a vector with CW's
}
```

For the second goal, we can use this snippet:

```
public void createSession(String from, InetAddress fromIP, ObjectOutputStream out)
{
    ChatWindow cw = getSession(from,fromIP);

    if (cw == null)
        myChatSessions.add(new ChatWindow(myUserName,myIP,from,fromIP,out));
    else
        cw.setVisible(true);
}
```

In this instance, when we want to create a session we first check to see if we can get it from the ChatAdministrator. If we can't, then we go ahead and create a whole new ChatWindow using our own IP, and the IP that represents the client which it should connect to on the other side. If we can, then make sure it is visible by setting visible to true.

The third goal of access to sessions or groups of sessions through the `getSession(..)` method. In this case, we pass in a vector containing the peers of the session we want. We then see if we can match this vector with one we have in the current session list like so:

```
public ChatWindow getSession(Vector peerInfo)
{
    Iterator chatIter = myChatSessions.iterator();

    while (chatIter.hasNext())
    {
```

```
        ChatWindow cw = (ChatWindow)chatIter.next();
        if (cw.equals(peerInfo))
            return cw;
    }

    System.out.println("Returning a null chat window!!!");
    return null;
}
```

The equals method in the ChatWindow class is overridden so that it can compare the two lists of peer objects to see if they are the same. EndSession works in much the same manner, except when we find the correct ChatWindow, we call `cw.close()` instead of returning `cw`.

5.3 Chat Application

So now that we have a way to keep track of all of our ChatWindows, how do we integrate this into the framework? This is the job of the ChatApplication class. The application class is simply an uppermost level of abstraction to allow for addition of code with less modification of old code. (See Section 4). Its main functionality is to take a nettable, see if it relates to our chatting feature, and if so, to tell the ChatFactory which it houses to work on it. This is accomplished in the following functions:

```
public boolean isType(Nettable n)
{
    return myChatFactory.matchesType(n);
}

public Application makeType(Nettable n)
{
    return new ChatApplication();
}

public void process(Nettable n)
{
    ChatInfoReceived ir =
        (ChatInfoReceived)myChatFactory.makeInfoReceived(n);

    ir.execute();
}
```

The ApplicationFactory at the highest level of the OOG framework actually makes the calls to `isType(...)` and `makeType(...)` in this instance, and then passes control to this application in the case that the `isType(...)` returns `true`.

5.4 Chat Factory

The ChatFactory uses the factory pattern to turn simple data it receives in the form of a Nettable into an executable InfoReceived object that actually performs the actions on the client side. It will then return this new object to the application, which then executes it. It works in two separate parts:

- Initializing the factory
- Checking for matches

This is how ours works:

```
//Initialization
private Vector myChatIRs;

public ChatFactory ()
{
    myChatIRs = new Vector();

    // Chat InfoReceived objects
    myChatIRs.add(new MultiChatInitInfoIR());
    myChatIRs.add(new MultiChatInitIR());
    myChatIRs.add(new MultiChatMessageIR());
    myChatIRs.add(new ChatCloseIR());

}

// Check to see if obj is a ChatNettable. If so,
//      ChatFactory should make InfoReceived
public boolean matchesType(Object obj)
{
    return ( obj instanceof ChatNettable );
}

//Searching for right parallel InfoReceived object
public InfoReceived makeInfoReceived(Nettable n)
{
    for ( int i = 0; i < myChatIRs.size(); i++ ) {
        ChatInfoReceived temp = (ChatInfoReceived)myChatIRs.get(i);

        if ( temp.isType(n) ) {
            ChatInfoReceived result = (ChatInfoReceived)temp.makeType(n);
            return (InfoReceived)result;
        }
    }

    // Should never get here becuase of matchesType(...) check
    System.out.println("WARNING: no match in chat factory IRs");
    return null;
}
```

5.5 Communication Architecture for Chat

Now that we have established the basic classes needed to make chat work on each individual machine, we need to examine how we will make chatting work from one machine to another. From above, we are aware of three separate steps in the chatting lifecycle: initializing chat connections, chatting by sending messages through those connections, ending the conversations by closing connections. To accomplish these

three basic functions, we use two separate base classes, the Nettable and the InfoReceived (see sections 3.3 and 3.4. The Nettable functions as a simple carrier of data, with no real logic built in and no capability of execution. The InfoReceived object takes the info from the Nettable and does some sort of action with that data on the receiving Client's machine.

5.5.1 Initializing Connections

The first step in the initialization process is the command. The Command serves to take input from the screen and figure out what to do with it. This is accomplished by sending this text to the client to which the console is connected, not the person who ran the actual p2p program. That client then processes the command, and relays some information back to the person who started the console. Thus, a command is really a subclass of InfoRecieved in that it indirectly takes data from the screen, and does some sort of operation on it. Our Chat Command would start like this:

```
public class ChatCommand extends Command
{
    //Used to check if this is the correct command
    private String commandName= "chat";

    public ChatCommand() {}

    protected ChatCommand(Nettable n) {
        super(n);
        commandInfo=n.toString();
    }
}
```

If one wishes to keep some information specific to the command, like when it was instantiated, this should be added to the initialization. Further the command will need to identify which command it is to the factory. This is accomplished by the first line of code specifying the `commandName`, coupled with the factory pattern in the `isType(...)` and `makeType(...)` functions which might look like this:

```
// Check to make sure it's the proper command
public boolean isType(Nettable n)
{
    StringTokenizer tokenizer = new StringTokenizer(n.toString(), " ");

    if (tokenizer.hasMoreTokens()) {
        String command = tokenizer.nextToken(); //takes first token
        return ( command.equals(commandName) );
    }

    return false;
}

// Instantiate the Command object
public InfoReceived makeType(Nettable n)
{
```

```
    return new MultiChatCommand(n);
}
```

The final important component is the `execute()` method, which in essence, ”does” the command. The execute might look like this:

```
public void execute()
{
    AllPeers peers = (AllPeers)p2p.getUtils().get(p2p.ALLPEERS);

    //Get The ConnectionAdministrator so we can figure out who to
    // send this nettable to
    ConnectionAdministrator ca =
        (ConnectionAdministrator)p2p.getUtils().get(p2p.CONNECTADMIN);

    //Get the correct connection (and socket)
    Connection ct = ca.getConnection(myNettableSender);
    Socket socket = (Socket)ct.client;

    // Get a Vector of Peer objects to chat with
    Vector peerConnectInfo = parsePeers(peers);

    // Add peer currently connected to the list
    InetAddress connectedToIP = socket.getInetAddress();
    peerConnectInfo.add( new Peer(peers.getPeerName(connectedToIP),connectedToIP) );

    //send the MultichatInitInfoNettable to the correct connection
    sendInitNettable(ct, peerConnectInfo);
}
```

The ConnectionAdministrator is called upon to find the connection to the IP address of the client who sent the original request i.e. the one with who typed the command in the console. In the `sendInitNettable(...)` function, this line of code is executed:

```
// "out" is the output stream
out.writeObject(new MultiChatInitInfoNettable(ct.client.getLocalAddress(),
                                              ct.client.getInetAddress(),
                                              peerInfo));
```

In effect this command sends a new MultichatInitInfoNettable to the input stream of the connection that is passed into the function. The client on the other side must then process this nettable. When the client receives the nettable, it will travel through the application factory, and end up in the ChatFactory. The ChatFactory is part of the application specific components and serves solely to convert the Nettable into an IR. To do so, the Nettable needs only a single accessor function that will be used to get the sent data when the new IR is initialized:

```
    public Object getContents()
    {
        return myPeerConnectInfo; //Vector of peer objects
    }
```

To check to see if the Nettable matches a certain type, the factory will use the `isType(...)` method of the corresponding InfoReceived object as follows:

```
    if ( temp.isType(n) )
    {
        ChatInfoReceived result = (ChatInfoReceived)temp.makeType(n);
        return (InfoReceived)result;
    }
```

The MultiChatInitInfoIR which will be created in the oogp2p.chat.IRs package, will contain the following code allowing for `isType(...)` and `makeType(...)` to be executed:

```
// Checks if proper parallel Nettable
public boolean isType(Nettable n)
{
    return (n instanceof MultiChatInitInfoNettable);
}

// Creates the InfoReceived object parallel to the Nettable
public InfoReceived makeType(Nettable n)
{
    return new MultiChatInfoInitIR((ChatNettable)n);
}
```

This code is fairly generic for all IR's in that the `isType(...)` will always determine if it's a nettable of the corresponding type, and the `makeType(...)` will always create an instantiation of itself using that nettable. This IR is then executed in the application, and will perform the following operations:

```
    ClientAdministrator clAdmin =
        (ClientAdministrator)p2p.getUtils().get(p2p.CLIENTADMIN);
    ChatAdministrator chatAdmin =
        (ChatAdministrator)p2p.getUtils().get(p2p.CHATADMIN);
    AllPeers peers = (AllPeers)p2p.getUtils().get(p2p.ALLPEERS);

    if (chatAdmin.getSession(myPeerConnectInfo) == null) {
        // Connect to any new peers
        connectToClients(clAdmin,peers);

        // Broadcast a chat initialization
        broadcastInitNettable(clAdmin);
    }
```

At this point we see the addition of the ClientAdministrator and ChatAdministrator. The ClientAdministrator acts as a Faade [1] to the underlying network model. It is used mostly to send Nettables to other clients that a certain computer knows about. In this case, the `connectToClients(...)` call will connect the current client to each of clients listed in the AllPeers object which contains all the peers we know about. Because we only add peers based on the text from the command line, this list should only contain the peers we typed into the console. The ChatAdministrator, as described above, is used to see if we are connected to the person we are attempting to chat with.

If not, then the function calls the `connectToClients(...)` function, and then calls the `broadcastInitNettable(...)` whose sole purpose is to tell the people we are trying to connect with that they should connect with us. This is accomplished by sending a MultiChatInitNettable to each client in list of peers, whose corresponding IR has functionality that is nearly the same except for the execute function which looks like this:

```
public void execute()
{
    ClientAdministrator clAdmin =
        (ClientAdministrator)p2p.getUtils().get(p2p.CLIENTADMIN);
    AllPeers peers = (AllPeers)p2p.getUtils().get(p2p.ALLPEERS);

    Peer me = new Peer(myNettableReceiver.getHostName(),
                       myNettableReceiver);

    // Connect to any new peers
    connectToClients(clAdmin, peers);
    createChatWindow(clAdmin);
}
```

Now, each other Client will connect to the list of peers the first client specified, along with the first client itself. So, we have a connection in each Client to each of other clients, which is recorded in the ChatAdmin. This provides us with the ability to send to each client that the ChatAdmin knows about individually, or as a group. The difference in this version of execute is the opening of the ChatWindow which as we have seen before provides for the graphical display of chat messages, as well as sending messages. The chat window allows users to actually send messages to other users.

The data transfer in the initialization phase will look like this:

5.5.2 Sending Messages to Connections

Now that the initialization process is complete, we are ready to send messages. To do so, we will be using the `sendMessage(...)` function of the chat window. When a user types a message in the window, and clicks the send button, a ChatMessageNettable will be created with the message as its payload:

```
    ChatMessageNettable messageNet =
        new ChatMessageNettable(myUserName, myConnectedToName,
                                myIP, myConnectedToIP, message);
```

The Nettable will then be sent through the output stream with which the window is created to another client. When it is received on the other side, the corresponding IR that is produced will execute the following code:

```
    ChatAdministrator chatAdmin =
        (ChatAdministrator)p2p.getUtils().get(p2p.CHATADMIN);
    ChatWindow cw = chatAdmin.getSession(myFrom, myFromIP);

    // Do some checks to make sure sender info is valid, etc.
    cw.addText(myFrom, myMessage);
```

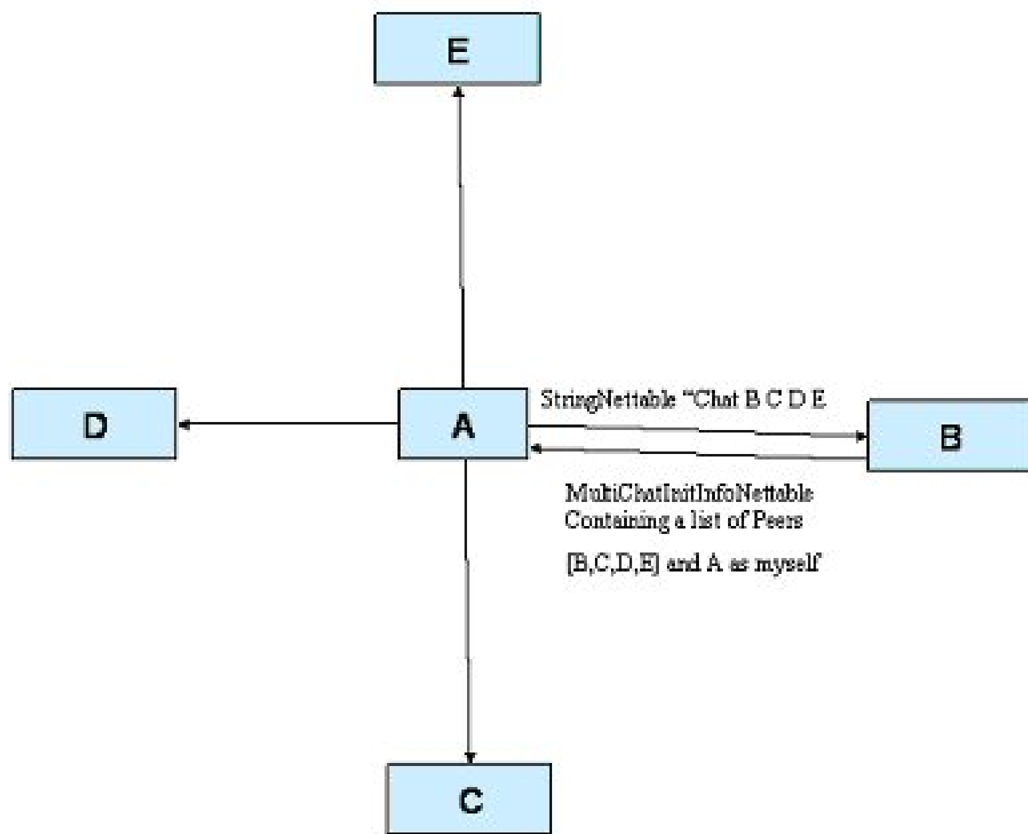

Figure 10: Obtaining Peer information before a chat initialization

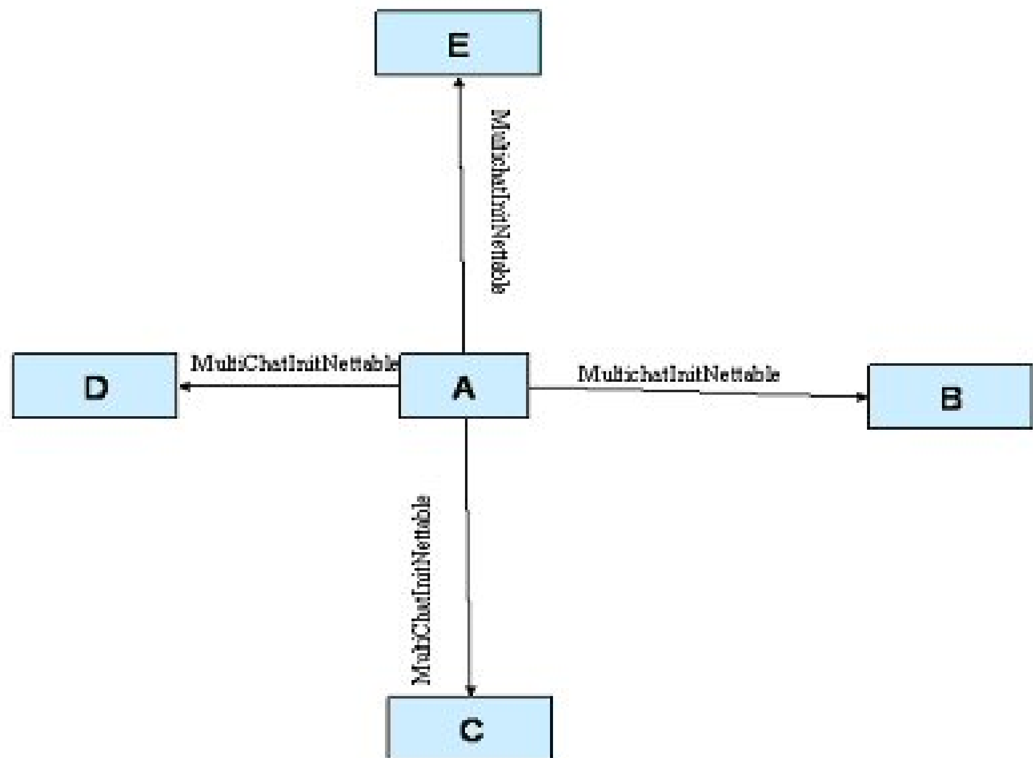

Figure 11: After each MultiChatInitNettable is received, each client opens a chat window

We use the chat administrator to figure out which window we should print the message to and then simply print that message using the `cw.addText()` function of the ChatWindow.

5.5.3 Closing Connections

Closing chat connections can be handled in one of two ways: we can manually close each connection or we can let the client clean up the connections after people end their sessions. In this case, it is more simple and just as effective to do the latter. If we were to implement this feature with manual closing, then all that would need to be added would be a listener that listened for the ChatWindow closing. When it closed, the listener would trigger an action that would send the ChatWindowClosedNettable (which might contain the IP of the closing ChatWindow) to the other connections that this ChatWindow contained. When the other ChatWindows received this Nettable they would replace the Nettable with an IR, extract the IP of the closed connection and simply close their own version of that connection.

5.6 Chat Class and Package Descriptions

A list with brief explanations and package information is as follows:

- **ChatAdministrator.java** - Used to find the correct chat window to send and receive data from. (`oogp2p.chat`)
- **ChatFactory.java** - Used to find which type of Nettable and thus the correct type of InfoRecieved to produce. (`oogp2p.chat`)
- **ChatWindow.java** - This class is used to display the actual contents of the chat conversation. Each Chat window must include an IP address in order to make for easier selection of correct windows. (`oogp2p.chat`)
- **ChatApplication** - An outer shell that houses the Factory class. When it receives a nettable it sees if its factory can process it. (`oogp2p.chat`)
- **MultiChatCommand** - Used to interpret user commands, start the chain of execution. (`oogp2p.chat.commands`)
- **ChatNettable** - Parent class for all Nettables in this package, has a ChatNettable-Parent Class for all nettables in this package, has a `fromIP` and a `toIP` used for determining where the nettable goes. The variables `fromIP` and `toIP` are used for determining where the Nettable is from and who its target is, respectively. (`oogp2p.chat.nettables`)
- **ChatInfoRecieved** - Parent class for all IR's in this package. (`oogp2p.chat.IRs`)
- **MultiChatInitInfoNettable** - Used to tell a client to send MultiChatInitNettables to certain people, so that the peers can start a chat (`oogp2p.chat.nettables`)
- **MultiChatInitInfoReceived** - Process peer information and broadcasts MultiChatInitNettables to all of them to initialize the chat. (`oogp2p.chat.IRs`)
- **MultiChatInitNettable** - Used to create IR that will initialize/begin the chat connection and open the chat window. (`oogp2p.chat.nettables`)
- **MultiChatInitInfoReceived** - Actually opens the ChatWindow and creates the connection. (`oogp2p.chat.IRs`)
- **MultiChatMessageNettable** - Used to carry the actual text message over the network (`oogp2p.chat.nettables`)

- **MultiChatMessageInfoReceived** - Used to display the message from corresponding Nettable in correct windows. (`oogp2p.chat.IRs`)

6 Future Work

Currently the GUI is a bare-bones Console Window. Future work will involve making GUI much more functional and user-friendly, through the use of good layout and good use of pull-down menus, buttons, etc.

There is a need for the generalization of both the input and output mechanisms. Currently the input and output are directly tied to the Console Window. This should be generalized so that future means of input and output and be developed for the system.

The processing of commands is a little bit odd in the OogP2P Framework. Under the current implementation, the output of the console window is directly tied to the client to which you are connectedthus the processing of the commands is done on that client. This is counterintuitive since commands should be processed locally.

References

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, Mass.: Addison-Wesley, 1995

[2] Shalloway, Allan and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2002.

[3] Stoica, Ion, et al. *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*. 2002

[4] Rowstron, Antony and Peter Druschel. *Pastry: Scalable, decentralized object location and routing for large-scall peer-to-peer systems.*