BSc (Hons) in Information Systems and Information Technology

Distributed Systems and Systems Integration

Report

Tomas Mikoliunas – D14123810

07/04/2016

Contents

1. Introduction

A lot of people uses different chat systems to talk with other people. The more common chat systems are like MSN Messenger [1], Skype [2] or Google Talk [3]. All of these require the user to create an account and sign in to be able to use their service. This require them to keep track of many different accounts to be able to talk to everyone they want to talk to. What if there existed a system that require no previously created account, no login and uses no central server to enable users to communicate with their friends? The application described in this report is an attempt to address this question.

For this assignment I have developed a project using Java RMI (Remote Method Invocation) based design showing an example of a Distributed System, which is a chat application. Java RMI lets us distribute Java objects instances across the network and different machines, and then we can invoke them from our local machine.

In this report techniques for developing a Peer-To-Peer chat system is studied and an analysis is performed of difficulties and problems for developing such a system. This Peer-to-Peer based chat application has been written in the Java language. Rather than routing messages through a centralized server, the chat server will be based loosely on a Gnutella design, describing how this method works. Specifically messages will be sent through the Peer-To-Peer network, but a Chat Server will be required to provide a newly started client with a list of known clients on the Chat Network.

2. Identifications of Challenges

Gnutella is a decentralized Peer-To-Peer (P2P) network that allows users to share files across the Internet without having to use a central server [4]. A Gnutella node (client) needs to know the location of at least one other Gnutella node in order to be able to work at all. Therefore a Gnutella node is initialised with a set of available nodes. A Gnutella node is connected to its working set and each connected node has its own working set of Gnutella nodes. No Gnutella node needs to have the same working set as another Gnutella node. This way of connecting creates a mesh of connected nodes (*Figure 1*).

Designing the Distributed Systems does not come easy. Some challenges need to be overcome in order to get the properly working system. The typical challenges in Distributed Systems are [5]:

- **Heterogeneity** – Describes a System consisting of multiple distinct components
- **Openness** – Property of each subsystem to be open for interaction with other systems
- **Security** – Confidentiality, Integration, Availability
- **Scalability** – As the system, number of resources, or users increase the performance of the system is not lost and remains effective in accomplishing its goals
- **Fault Handling** – Some measurements must be taken: Detect failures, Mask failures, Recover from failures, Build redundancy

- **Concurrency** – Concurrency issues arise when several clients attempt to request shared resource at the same time

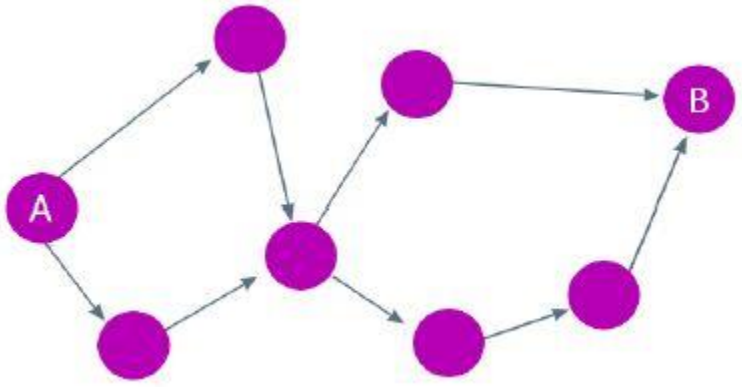

Figure 1. Gnutella Design

Creating a simple chat system using a client/server model is very easy, and can be done in a less time if the programmer is used to network programming. All that is needed is a server application which can supply the location of a specific user to a client requesting that user and a client application that will connect to a given location for a user.

My challenge was to create a chat application with a Peer-To-Peer Distributed System model, which is more difficult to do. The main issue with such an application is the fact that the location for anyone (Chat clients) is simply not known. A registration server (Chat server) must be available to store information of instances where chat clients are connected. Distribution requires interaction between applications, and the challenge I have had to overcome that if the central server 'dies' the chat clients still would be able to communicate between them.

When computers communicate over networks, there exists the potential for communication problems. For example, server computer could malfunction, or a network resource could malfunction. If a communication problem occurs during a remote method call, the exception will be thrown which must be handled in the graceful way with appropriate message to the user, explaining what has happened. I had to follow the requirements like: consistence of data, maintainability, fault tolerance, load balancing.

Also the challenge I had to overcome was to make sure that every communicating client processes the new messages only once and to make sure they shares the list of references of 'known clients' on the network. I found difficult and confusing to implement it to work correctly as my goal was to make it possible to participate any number of clients in communication simultaneously.

Distributed System needs methods to communicate. By reading a tutorials and watching a videos, I have found out that those problems can be addressed by implementing and extending required interfaces and classes, writing significant code and RMI based technology solves mostly all of the challenges, which has a features like method calls between different virtual machines etc. RMI provides the call of the methods on other virtual machines that are located on the same host like the caller or on machines that are connected over network.

All those challenges can be achieved by distributing the application over 2 or more hosts (chat clients). The central chat server has been implemented which saves and provides the data required in order for the chat clients be able to communicate between them directly.

To test a system a server will be started on the local machine, then as many clients as possible will be started to check if a server is a limited to particular amount of clients. The limiting factor of the test could be the number of threads that could be started on the machine. The limitation of the server or the client could be the maximum number of connections that the operating system allows and the linked hash table where the clients are saved would lead to problems, because the access of a linked hash table is not very fast, and for each message that is sent from one particular client to the other participating clients, that client has to iterate through the whole table.

3. Design Description

This Java-project realizes a chat system by using Java RMI (Remote Method Invocation) design. RMI allows the invocation of methods on remote objects that reside on different Java Virtual Machines (JVM). The Project consists of a server and a client executable source codes.

The first step in creating a distributed application with RMI is to define the remote interface that describes '***remote methods***' (*Figure 2*) through which the clients will interact with the server and other remote clients using RMI. To create a remote interface, we need to define an interface that extends interface ***'java.rmi.Remote'***. Interface '**Remote**' is a '*tagging interface'* – it is not declare any methods. An object of a class that implements interface '**Remote**' directly or indirectly is a *remote object* and can be accessed from any Java virtual machine that has connection to the computer on which the remote object executes. A remote object must implement all methods declared in its remote interface.

If a communication problem occurs during a remote method call, the remote method throws a '**RemoteException**' which is a checked exception. So, each method in a '**Remote**' interface must have *throws* clause that indicates that method can throw **'RemoteException'** (*Figure 2*):

```
public interface ChatServerInterface extends Remote {

    // register chat clients to server
    public void registerChatClient(String clientName, ChatClientInterface rmiClient) throws RemoteException;

    /**
     * Request a set of clients from the chat server.
     */
    public abstract Map getClientsList() throws RemoteException;
}
```

Figure 2. Remote Chat Server interface

Class '**UnicastRemoteObject**' (extended from 'FXMLChatServerController.java' line – 42, 'FXMLChatClientController.java' line – 34') provides the basic functionality required for all remote objects, its constructors exports the object to make it available to receive remote calls. Exporting the object enables the remote object to wait for another remote object connections. This enables the object

perform *unicast communication* (point-to-point communication between two objects via method calls) using standard stream-based socket connections.

In this RMI based application, before the server becomes remotely accessible to clients, it must be registered in the registry, a process called "binding". We need specify the IP address in the top field of GUI panel and press '**Start Server**' button (*Figure 3*). The registry runs on the server where the remote object exists. The server will hold a list of references (the names will be shown in the GUI panel), to all remote clients who have registered with this server on the network, and they will be accessible to other clients through RMI (*Figure 3*).

Figure 3. Server registration to registry and hold a list of client references

The registry of the server is queried by the client by specifying the URL of the remote server (*Figure 4*). If the IP address entered does not match the remote server registry IP address, client gets an error message (*Figure 5*). Chat clients must also provide their names to be able to register with the central server. Every client refers to the remote Chat server through '**ChatServerInterface**' (*Figure 2*) – the remote interface for the remote Chat server implementation. The client can use this interface as if it referred to a local object running in the same virtual machine. When a client successfully connects/registers to the central server, he gets a list of peers ('known clients') which contains the IP addresses of them and it will allow to create a link between them and establish a connection (*Figure 4*). After this initialization, the client can broadcast messages via RMI to its peers and receive messages via RMI calls from its peers, at this point the client turns into a "server" able to send a messages, share a list of available clients and also can accept the new connections coming in from other clients (*Figure 4*). On receiving every new message client gets updated his list with the list of 'known clients' on the network from other clients who have their own list of 'known clients', because every clients when sends a new message he shares the list of references also.

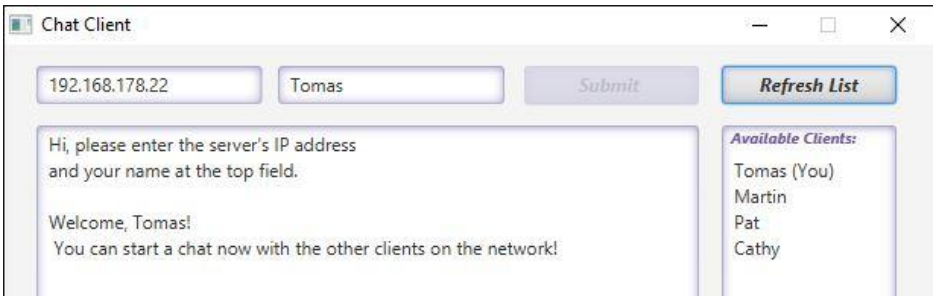

Figure 4. Successfully started client

Every new message sent by the client is distributed by the other receiving clients by their list of ‘known clients’. For example, we have got 3 or more clients on the network. Can the client ‘1’ talk with the client ‘2’, which got the information of the client ‘3’? Yes. Then the client ‘2’ will send the information received from the client ‘1’ to the client ‘3’, and also will share his list of ‘known clients’, so the client ‘3’ will know about the client ‘1’ from this point of time and also will be able to send a new messages directly to the

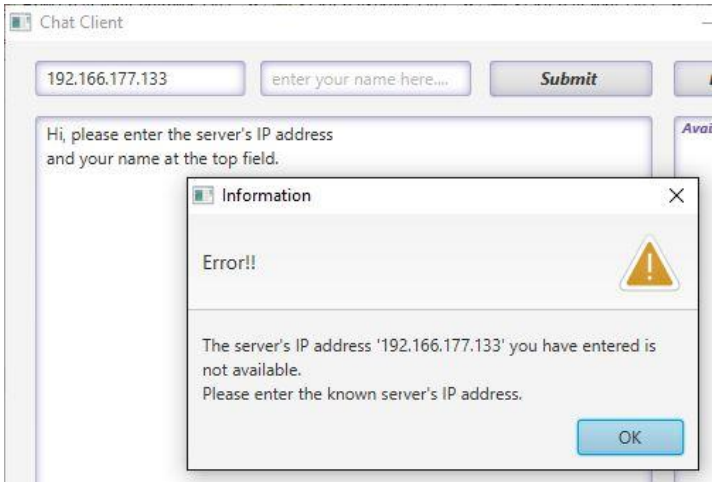

Figure 5. Remote server IP address does not match

Client ‘1’ (*Figure 6*). As we can see from the (*Figure 6*) the messages and the list of references will be distributed to all ‘known clients’.

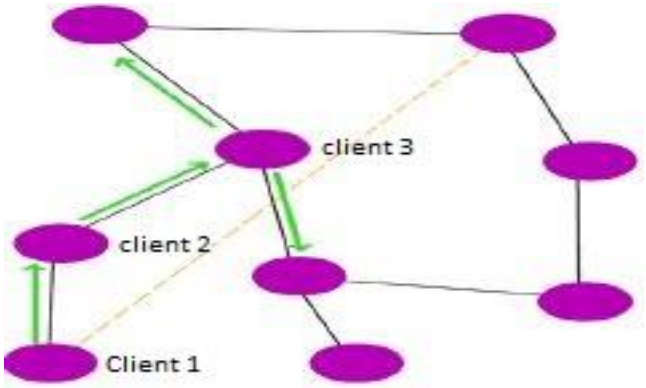

Figure 6. Distributed System example

Therefore, once a chat client made the initial connections to the chat server and downloaded the list of ‘known clients’, it no longer depends on the server for either keeping this list updated or sending and receiving messages. Active chat client application will manage the ‘known clients’ list updates based on the data received from the other clients.

The Chat client can also update his 'known clients' list by pressing the button '**Refresh List**' in GUI panel and as a result he will get the updated list of 'known clients' from the central server.

On receiving a new message, the Chat client will determine if this message has never been seen before and also will determine if the new clients exist on the network who has not seen this message. It will be done by comparing his current 'known clients' list with received list from the other client who has sent the message. By comparing these lists of 'known clients' references, will be determined if that message has been already processed to particular clients, and if so has happened, it will be discarded to send it again (*Figure 7*). Oher client does the same procedure, so as a result every 'known client' processes the new message only once.

Every new message will contain information of:

- Username of client who has sent the message
- The content of the message (String of text)
- The list of 'known clients' which have already seen the message

```
// receive new message from other clients, including list of clients who has seen this message
@Override
public void receiveMessage(Map<String, ChatClientInterface> clientsList, String message) throws RemoteException {

    displayMessage(message); // received message displayed to this user, then passed on for iteration of known clients

    // add only new clients instances received
    this.clientsList.putAll(clientsList);

    // after updated clients list, refresh clients list on the window
    displayClientsNames(this.clientsList);

    for (ChatClientInterface client : this.clientsList.values()) { // 'this' refers to this user current client list

        if (!(clientsList).containsValue(client)) { // if message not seen by other clients, send this message to them

            // send message to other clients 'receiveMessage' method, include updated 'clientsList'
            client.receiveMessage(this.clientsList, message);
        }
```

Figure 7. Process the message if it has never been seen before

The client must implement public method to send response to central server on request, and several public methods to communicate with other clients (*Figure 8*).

As shown in the remote chat client interface (*Figure 8*), clients must implement the '**isAlive()**' method, that can be used by the central server to determine whether a remote client is still alive.

```
public interface ChatClientInterface extends Remote{

    // display the messages for each client on the network
    public void displayMessage(String message) throws RemoteException;

    // receive new message from other clients, including list of clients who has seen this message
    public void receiveMessage(Map<String, ChatClientInterface>clientsList, String message)throws RemoteException;

    // check if whether the clients are alive on the network
    public boolean isAlive() throws RemoteException;
}
```

Figure 8. Remote Chat Client interface

In this system clients can join and leave ('die') independent of all other clients at any time. To leave the Chat communication client needs to press red button at the top-right corner in the GUI panel. The central server will periodically verify whether clients is still reachable by calling '**isAlive()**' method from Chat client interface (*Figure 8*). The central server will make this verification every 2 seconds. The central server will find out that particular client is not alive if a calling '**isAlive()**' method fails (i.e. throws a '**RemoteException**'), meaning that the reference no longer pointing to any chat clients. Then that client will be removed from the 'known clients' list. The central server is also able to supply clients with a of all known chat clients when requested, so the central server must implement '**getClientsList()**' method (*Figure 2*), which can be called by the client at any given time by pressing '**Refresh List**' button (*Figure 4*).

Once the

This RMI based design application can be run on multiple separate machines on LAN (Local Area Network) or on the same machine. I went for RMI based application because it gives a bunch of advantages like multithreading, distributed garbage collection, feature to collect remote server objects that are no longer referenced by any clients on the network, object marshalling etc.

Also my choice for RMI was because it is Java language based application and it can be run on any platform and it is "Write Once, Run Anywhere" approach.

4. Issues

The problem to watch out for is to make sure a client does not forward a message it has already seen. That means all clients must have some unique identifier, e.g., the originating client name which is stored in '**LinkedHashMap**' of type '**Map**' collection ('FXMLChatClientController.java' line - 156) and used as an ID, and a client must check each incoming message included with the list of clients who already have seen this message against the list of clients what this "current client" has. So, it was very important to avoid a sent messages replication.

Also the place where I would like to spend more time exploring, is the management of the peers 'known clients' on the chat client side when a peer has some network related issues or leaves the chat, so the other clients would be able to discover that without help of central server. Implementing such discovery mechanism though might change the overall application design to pure peer-to-peer architecture.

5. Evaluation

As a result of evaluation the whole system has been created from scratch that uses the advantages of RMI. Benefits from this design technique is that the code can be written and tested on a single machine in a single process, JDK-compatible debugger can be used for debugging both server and client, this RMI based chat system can be demonstrated on a laptop without changing any network or TCP/IP settings and also there is no need to run an RMI registry at the time of debugging.

I have implemented Peer-To-Peer communication mechanism in order to get away from the drawbacks a client/server model suffers from. The following points are a few of the drawbacks that the classic client/server model suffer from:

- It has a single point of failure. Essentially this means that if the server goes down the service is not able to function properly if it is even able to function at all.

- The server need to serve a possibly high amount of work because all requests in the system is directed to the server, so the load of the server is high.

This Peer-To-Peer chat application is interesting and potentially effective mainly because it takes away the single point of failure drawback that the client/server application suffers from and it gives an application a distributed communication instead. This enables peers to communicate directly with each other instead of having a server in the middle which all communication passes through. This makes it possible for a properly implemented Peer-To-Peer application to tolerate loss of multiple nodes and each node will possibly perform a balanced amount of work (in relation to the total amount of work for all nodes).

Compare this with a client/server application that has a single central server, meaning that if the server goes offline it is not possible to use the application at all. The central server will have a high load of work to be performed since it will need to serve possibly all requests, while in this Peer-To-Peer RMI based design chat application each node will serve a portion of the total amount of requests since the work is distributed on each node.

By comparing these application designs and specifications, we can see that my created RMI has strengths over basic client/server:

- If central server fails to work all the other clients connected to network still continue to work

- All the resources and contents are shared by all the peers, unlike server-client architecture where Server shares all the contents and resources

- Each computer in such a network is part server and part client. This means that each computer on the network is free to share its own resources

Also the strength of my system is that can be created any number of the clients as long as the operating system allows and the linked hash table capable to manage where the references to the chat clients are saved. All these instances for the clients can participate in the chat process simultaneously.

My RMI system will work well on different separate machines on Local Area Network or it can be used for a communication on the same (local) machine. This RMI design based chat application would be suited for multiple communication processes like sending messages Strings in the companies between a local staff, in college campus etc.

However, the weaknesses of my system could be when the large number of clients connected at the same time, we could have a latency issue, the delay from input and output received by the client, since every message sent is appended with the list of all 'known clients' to the sending chat clients, and also because every remote invocation from the client results into a new Thread and a new connection.

Comparing to the other modern chat systems [1], [2], [3] this RMI chat system does not allow users to create their own chat "rooms", here is no possibility to change the current user name without logging off (leaving the chat communication). To communicate only 'String' are send between Chat clients and no other data format could be exchanged.

Another area of weaknesses is chat client peers discovery mechanism, for example, if a new chat client peer comes online and the server is unavailable, this peer will not be able to participate in the chat communication, because that chat client will not be able to detect the 'known clients' locations on the network without the help of central server.

My RMI system is not suited for use on non-LAN type networks, so it will not work on WAN (Wide Area Network) connection. Also RMI is Java language-only, the distributed objects have to be implemented in Java. This could be problem when interfacing with other languages applications.

6. Conclusions

RMI is a very good way to implement Distributed Systems. The objects can be distributed dependent on runtime. The protocol of the communication is checked by Java compiler, so the programmer does not have to care about this.

My RMI Chat application design has its strengths and weaknesses. Depending on the environment where this application is going to be used, it might impose some scalability issues in the form of increased network traffic coming to and generated by the participating peers. Some improvements can be made to the application design to reduce the network overhead.

Therefore, the possibilities of the chat system can be enhanced. The chat systems, for example, like modern chat applications [1], [2], [3] mentioned before allow users to create own "rooms" and to change their names without logging out. After this system is running those methods can be added at later time.

7. References

1. MSN Messenger [Online]. Available from: https://en.wikipedia.org/wiki/Windows_Live_Messenger [Accessed on: 24 March 2016].
2. Skype [Online]. Available from: https://support.skype.com/en/faq/FA10983/what-are-p2p-communications [Accessed on: 24 March 2016].
3. Google Talk [Online]. Available from: https://en.wikipedia.org/wiki/Google_Talk [Accessed on: 24 March 2016].
4. Gnutella [Online]. Available from: http://whatis.techtarget.com/definition/Gnutella [Accessed on: 24 March 2016].
5. Martin McHugh (2016) *Distributed Systems and Systems Integration*. Lecture notes.