AT70.05: Computer Networks
August 2015 Semester

Community Chat System (CCS)

Submitted by:
Worapol Leerunyakul (st116391)
Sariya Tangthamniyom (st116401)
Teera Kritpranam (st117895)
Delani Rushanka Perera (st117810)

Submitted to:
Prof. Kanchana Kanchanasut
Dr. Apinun Tunpan

Department of Computer Science and Information Technology
School of Engineering & Technology, AIT
Asian Institute of Technology
Thailand
November 2015

Acknowledgement

We would like to take this opportunity to thank all the people those who helped us during the implementation of the project in order to obtain this success.

Our sincere gratitude goes out to:

Prof Kanchana Kanchanasut of the Asian Institute of Technology, the instructor of the Computer Network course and also the project supervisor, for her initial encouragement to carry out this project in this area and great guidance towards the project and for her valued time.

Dr. Apinun Tunpan of the Asian Institute of Technology, who helped us to learn a new programming language, Python and also providing his great valuable guidance during the project implementation period.

Our colleagues who helped and encouraged to make this project a success.

Table of Contents

Chapter 1
Introduction

There are numerous products available that allow for real time "chatting" over the Internet. The purpose of this project is to implement a chat application that will allow users with an internet connection to engage in public conversations. The development of this project centered on the development of a message protocol that would allow the application to properly log in users, send messages, and perform system maintenance.

1.1 Background

This project is to create a peer to peer chat application chat with many other clients in the same common chat group. This project is to simulate the multicast chatting. In the case of multicasting when a message is sent to a group of clients, then only a single message is sent to the router. The main purpose of this project is to provide multi chatting functionality through network.

1.2 Objectives

- Design a chat room system to peer-to-peer network.
- Develop a log file showing the actions / decisions that have occurred when the router is powered up , sorted according time
- Develop a method to sync and merge the message among users and also when router go down develop a method to sync the history message

1.3 Problem Statement

When developing community chat system , the arise problems were ,

1. several chat room distributed among the router
2. distributed server
3. Merging and sorting
4. Maximum size of the message that user can send over the network
5. error recovery

Only few of the above mentioned problems could be solved by the system .From further analysis we could find methods to solve although we could not implemented in the system .

1.4 Assumptions

The system is supported only for text communication.

Chapter 2
Literature review

Online chat system

Online chat is one kind of communication over the Internet that offers a real-time transmission of text messages from sender to receiver. Chat messages are generally short in order to enable other participants to respond quickly. Therefore, a system that is similar to a real-time conversation is created, which distinguishes chatting from other text-based online communication forms such as Internet forums and email. Online chat may address point-to-point communications as well as multicast communications from one sender to many receivers and voice and video chat, or may be a feature of a web conferencing service. It includes web-based applications that allow communication – often directly addressed, but anonymous between users in a multi-user environment. Web conferencing is a more specific online service, that is often sold as a service, hosted on a web server controlled by the vendor.

User authentication

User authentication has originally been used to verify user identity before accessing to databases. As the Internet keeps growing in size, advancement and complexity, various authentication approaches such as one-time password, secure-shell(ssh) and biometric authentication have been developed for security purposes in the network. User authentication has been integrated to chatting system for more than 15 years. The essential point of integrating authentication to chatting system is to identify ownerships on messages that have been sent by particular users in the system. MSN messenger, AOL instant messenger and Facebook chat are examples of well-known chatting systems with at least has basic level of user authentication. The latter system also offers security management for users to choose from such as login approvals, trusted machines and login alerts.

Distributed computing

Distributed computing system is a software system in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal. Three significant failures of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components. Examples of distributed systems vary from SOA-based systems to massively multiplayer online games to peer-to-peer applications.

Peer-to-Peer Architecture

The term peer-to-peer is used to describe distributed systems in which labour is divided among all the components of the system. All the computers send and receive data, and they all contribute some processing power and memory to a distributed computation. As a distributed system increases in size, its capacity of computational resources increases.

Wireless Ad hoc network

A wireless Ad hoc network is a decentralized type of wireless network. The network is ad hoc because it does not rely on a pre existing infrastructure, such as routers in wired networks or access points in managed wireless networks. Instead, each node participates in routing by forwarding data for other nodes, so the determination of which nodes forward data is made dynamically on the basis of network connectivity. In addition to the classic routing, ad hoc networks can use flooding for forwarding data.

Wireless mobile ad hoc networks are self-configuring, dynamic networks in which nodes are free to move. Wireless networks lack the complexities of infrastructure setup and administration, enabling devices to create and join networks "on the fly" - anywhere, anytime.

Multicast

Multicast is group communication where information is addressed to a group of destination computers simultaneously. Group communication may either be application layer multicast or network assisted multicast, where the latter makes it possible for the source to efficiently send to the group in a single transmission. Copies are automatically created in other network elements, such as routers, switches and cellular network base stations, but only to network segments that currently contain members of the group.

Network assisted multicast may be implemented at the Internet layer using IP multicast, which is often employed in IP applications of streaming media, such as Internet television scheduled content and multipoint videoconferencing, but also for ghost distribution of backup disk images to multiple computers simultaneously. In IP multicast the implementation of the multicast concept occurs at the IP routing level, where routers create optimal distribution paths for datagrams sent to a multicast destination address.

Data Synchronization

Data synchronization is the process of maintaining the consistency and uniformity of data instances across all consuming applications and storing devices. It ensures that the same copy or version of data is used in all devices - from source to destination. The process is implemented in distributed systems where data elements are routed between several computers or systems. Each computer may modify original data versions, depending on requirements. it is also used in data mirroring, where each data set is exactly replicated or synchronized within another device.

Merge Algorithm

Merge algorithms generally run in time proportional to the sum of the lengths of the lists; merge algorithms that operate on large numbers of lists at once will multiply the sum of the lengths of the lists by the time to figure out which of the pointers points to the lowest item, which can be accomplished with a heap-based priority queue in O(log n) time, for O(m log n) time, where n is the number of lists being merged and m is the sum of the lengths of the lists. When merging two lists of length m, there is a lower bound of 2m-1 comparisons required in the worst case.

The classic merge outputs the data item with the lowest key at each step; given some sorted lists, it produces a sorted list containing all the elements in any of the input lists, and it does so in time proportional to the sum of the lengths of the input lists.

Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human readable output.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

Maximum transmission unit

Maximum transmission unit (MTU) of a communications protocol of a layer is the size of the largest protocol data unit that the layer can pass onwards. MTU parameters usually appear in

association with a communications interface. Standards, Ethernet, for example, can fix the size of an MTU or systems such as point-to-point serial links may decide MTU at connect time.
A larger MTU brings greater efficiency because each network packet carries more user data while protocol overheads, such as headers or underlying per-packet delays, remain fixed; the resulting higher efficiency means an improvement in bulk protocol throughput. A larger MTU also means processing of fewer packets for the same amount of data. In some systems, per-packet-processing can be a critical performance limitation.
However, this gain is not without a downside. Large packets occupy a slow link for more time than a smaller packet, causing greater delays to subsequent packets, and increasing lag and minimum latency. For example, a 1500-byte packet, the largest allowed by Ethernet at the network layer, ties up a 14.4k modem for about one second.

Large packets are also problematic in the presence of communications errors. Corruption of a single bit in a packet requires that the entire packet be retransmitted. At a given bit error rate, larger packets are more likely to be corrupt. Their greater payload makes retransmissions of larger packets take longer. Despite the negative effects on retransmission duration, large packets can still have a net positive effect on end-to-end TCP performance.

Chapter 3
Methodology and System Design

According to the objectives , our system contain 4 chat rooms. User can login to the system by entering the username and password to authentication and after that the chat room number (1,2,3,4). After log in to the system user can see the synchronized message and user can open and see the log file which has been stored in the router . The message will be send with the IP address and the current chat room no of each user as an identification. To send the message over the ad-hoc network we used the multicast method . Other users in different chat rooms would not be able to see the message until they log to the appropriate chat room.

When one router goes down , and when it connects to the network router will send message by asking about the log file history from other routers routers and synchronize the chat history file

In this project we used the Optimized Link State Routing Protocol (OLSR), which an IP routing protocol optimized for mobile ad hoc networks.

Network Design

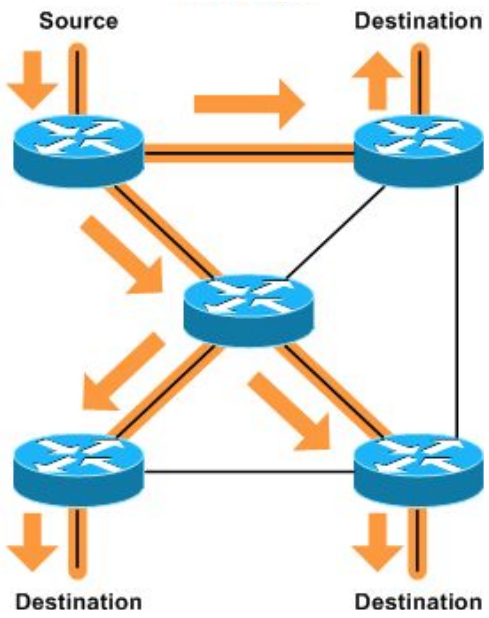

Figure 3.1 : Network Design for multicast

Flow chart of Community Chat System (CCS)

Please see the attached file names ***"Flow chart of CSS.html"*** in our project's unzip file.

3.1 FSM Diagram

Scenario 1 - state diagram for one client connecting to chat room

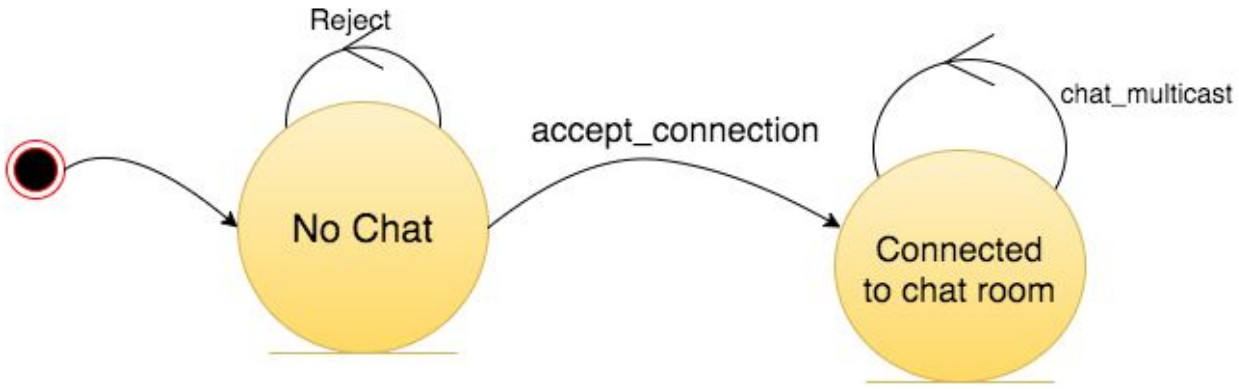

Figure 3.2 : State Diagram for one cient

Scenario 2 - state diagram for a chat room

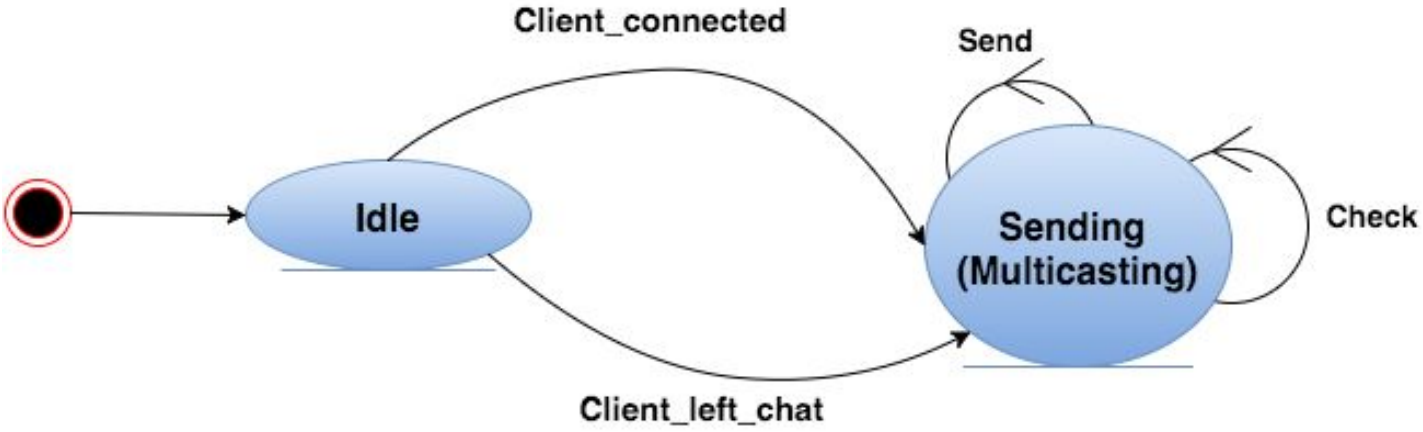

Figure 3.3 : State Diagram for a chat room

Chapter 4
Implementation/Analysis and Design

4.1 Multicasting

One of the challenges is to send messages over multihop ad-hoc network. Therefore, we cannot apply Broadcasting since the process can achieve only one hop.

With Multicasting mechanism, data is going to be relayed to many nodes in the network without messages flooding back to the origins. To apply this technique, first, it is necessary to define a multicast group, the IP beginning with "**224.xxx.xxx.xxx**". Then we will specify a multicast port. We can also tell the nodes how many hops the messages can be passed by stating a number in ? into this line of code "**s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ?)**". In this example, we need 2 hops.

Here is the setup:

```
MCAST_GRP = '224.1.1.1'
MCAST_PORT = 8888
addressInNetwork = []

send_address = (MCAST_GRP, MCAST_PORT) # Set the address to send to
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)    # Create Datagram Socket (UDP)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # Make Socket Reusable
s.setblocking(False) # Set socket to non-blocking mode
s.bind((MCAST_GRP, MCAST_PORT))
mreq = struct.pack("4sl", socket.inet_aton(MCAST_GRP), socket.INADDR_ANY)
s.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
s.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)
```

Figure 4.1: Multicast setup

Then we need to configure our Multicast Group IP to the router:

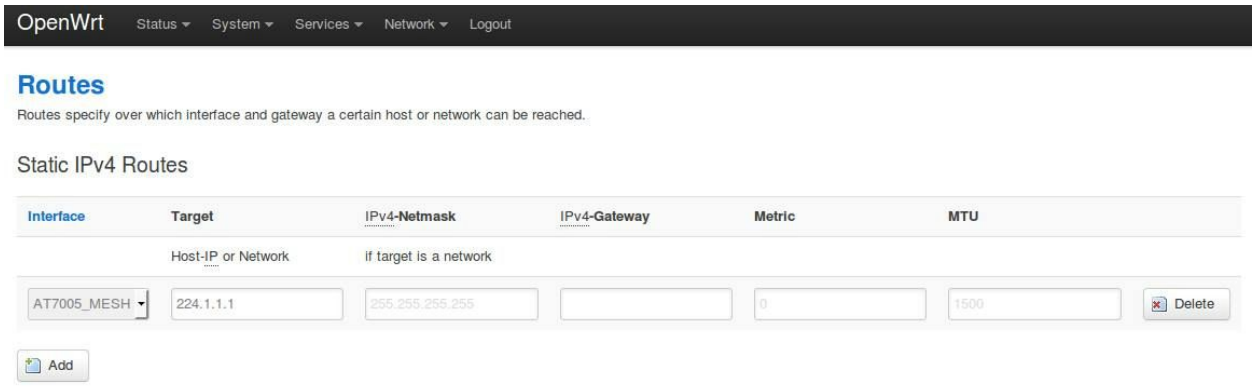

Figure 4.2: Multicast configuration

4.2 Message synchronization

Input & Output

In the beginning, we used "**raw_input('-')**" for getting inputs. Unexpectedly, while we were typing or waiting, messages sent from the others were lost because the process was blocking at this line of code. Thus, we apply "**select**" function to handle outgoing and also incoming messages. With "**select**", now we can send and receive messages simultaneously.

```
def getLine():
    inputReady,outputReady,exceptionReady = select.select([sys.stdin],[],[],0.0001)
    for socketSelect in inputReady:
        if socketSelect == sys.stdin:
            input = sys.stdin.readline()
            return input

    return False
```

```
input = getLine()
if input :
    localtime = time.asctime(time.localtime(time.time()))
    s.sendto(localtime+"/"+roomNumber+"/"+input, send_address)
```

Figure 4.3: Implementation of how to input & output messages

Data storage

To store data, our design is to use four text files since the routers have only 32MB of RAM, so any Database applications are not suitable. In addition, every routers are able to receive messages as we use Multicasting mechanism. Therefore, when messages are arrived at destinations, they will be written into their log file regarding to the chat room number.

Furthermore, while users are using a particular chat room, the messages which are presented should be only for that specific room but messages sent for the other chat rooms ought to be kept as well. Hence, we have to handle this.

Here is the code:

```
message = message.split("/")

RCVTime  = message[0]
RCVRoom  = message[1]
RCVInput = message[2]

f1 = open('log1.txt', 'a') ###
f2 = open('log2.txt', 'a') ###
f3 = open('log3.txt', 'a') ###
f4 = open('log4.txt', 'a') ###

if RCVRoom == "1" :
    f1.write('%s|%s|%s|%s'.rstrip('\n') %(address,RCVTime,RCVRoom,RCVInput))
elif RCVRoom == "2" :
    f2.write('%s|%s|%s|%s'.rstrip('\n') %(address,RCVTime,RCVRoom,RCVInput))
elif RCVRoom == "3" :
    f3.write('%s|%s|%s|%s'.rstrip('\n') %(address,RCVTime,RCVRoom,RCVInput))
elif RCVRoom == "4" :
    f4.write('%s|%s|%s|%s'.rstrip('\n') %(address,RCVTime,RCVRoom,RCVInput))

f1.close()
f2.close()
f3.close()
f4.close()

if RCVInput and roomNumber == RCVRoom:
    print address , RCVTime , RCVRoom , RCVInput
```

Figure 4.4: Implementation of how to handle data storage

4.3 User Authentication

Introduction

Accessing to ad-hoc chat service requires user authentication. All users must sign up for and sign in with their own accounts. Since the purpose of this project is to implement and mainly focus on the functionalities of ad-hoc chatting system, basic user verification has been applied only to identify the ownerships of messages in chat system. Once the program is executed, visitors will be questioned whether they have their own accounts for logging into the chat system as shown in Figure 4.5. Other answers rather "Yes" and "No" will not be accepted by the program.

```
def signin():
     while True:
        newuserfile     = 'newuser.txt'
        registered = raw_input("Do you have an account? [Yes/No]")

        if registered.lower() == "yes":
                while True:
                        id = raw_input("Username:")
                        pw = raw_input("Password:")
                        userlist = open('userlist.txt', 'r')
                        loginuser = id+"||!^"+pw
                        islogin = userlist.read()
                        checklogin = islogin.split("\n")
                        for index1 in range(len(checklogin)):
                                if loginuser == checklogin[index1]:
                                        #print loginuser
                                        logginguser = loginuser.split("||!^")
                                        loguser = logginguser[0]
                                        print "Logged in as: %s" % (loguser)
                                        return loguser
                        print "Username or password is invalid"

        elif registered.lower() == "no":
                signup()
```

Figure 4.5: python code for sign in. The program will take input from user, lowercase the input and check if the input match to "yes" or "no" before executing another line of code.

```
root@N20_ApinunTunpan:~# python main8.py
Accepting connections on port 0x22b8
Do you have an account? [Yes/No]skskskskskskskks
Do you have an account? [Yes/No]
```

Figure 4.6: an initial question for visitors. Only "Yes" and "No" are accepted by the program

According to figure 4.6, a visitor attempts to answer the question with “sksksksksksksksks” which is not accepted by the program. The program hence does not proceed to the next step, return the initial question and wait for the visitor to answer “Yes” or “No”.

Sign up

Sign up messages will appear once visitors answer “No” to the initial question. Visitors will be asked to create their accounts. Username, password and password confirmation must be entered to the program. There are 3 possible scenarios for this step as shown below.

1. Visitors enter a username which has already been taken
2. Visitors enter a unique username but password does not match with password confirmation
3. Visitors enter a unique username and password matches with password confirmation

Python code for sign up is shown in Figure 4.7.

```
def signup():
    while True:
        newid = raw_input("Set your username:")
        newpwd = raw_input("Your password:")
        chkpwd = raw_input("Password confirmation:")
        signupdelimeter = "||!^"
        if newpwd == chkpwd:
                userlist = open('userlist.txt','r')
                existuser = newid+signupdelimeter
                content = userlist.read()

                if existuser in content:
                        print "This username has already been taken"

                else:
                        userlist.close()
                        userlist = open('userlist.txt','a')
                        insertuser = newid+signupdelimeter+newpwd
                        userlist.write('%s' %(insertuser))
                        userlist.write('\n')

                        userlist.close()

                        newuser = open('newuser.txt','a')
                        newuser.write('%s' %(insertuser))
                        newuser.write('\n')
                        newuser.close()

                        print "Your account has been created"
                        return
        else:
                print "Password does not match"
```

Figure 4.7: Python code for sign up.

In first scenario, the program will reply a message “This username has already been taken”. Visitors will be asked to create their accounts with different username as shown in figure 4.8.

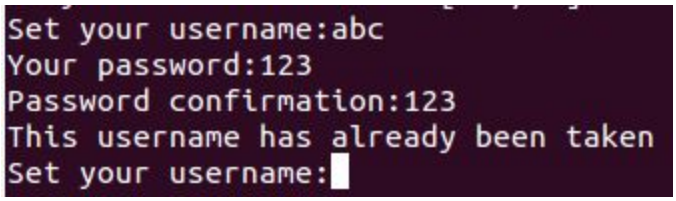

Figure 4.8: Response messages for sign up scenario 1.

In second scenario, the program will reply a message “Password does not match”.Visitors will be asked to create their accounts again as shown in figure 4.9.

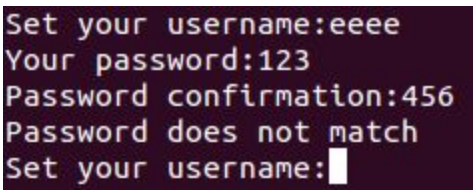

Figure 4.9: Response messages for sign up scenario 2.

In third scenario, the program will reply a message “Your account has been created”. Visitors will be directed to the initial question as shown in figure 4.10. Their accounts are ready to use.

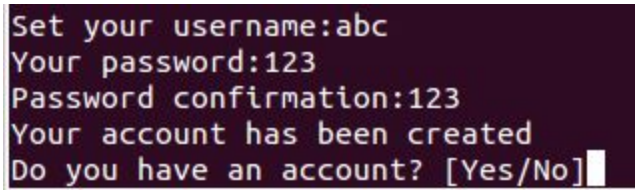

Figure 4.10: Response messages for sign up scenario 3.

Username and password which has been created will be stored in the ‘userlist.txt’ file in the following manner; username+delimiter+password. In this project, “||!^” has been used as a delimiter. An example of ‘userlist.txt’ is shown in figure 4.11

```
 GNU nano 2.3.6                                File: userlist.txt

may||!^123
teeramoo||!^1
niew||!^123
t||!^1
delani||!^678
moo2||!^123
```

Figure 4.11: Examples of 'userlist.txt'

Sign in

Sign in messages will appear once visitors answer "Yes" to the initial question. Visitors will be asked to enter their username and password. There are 3 possible scenarios for this step as shown below.

1. Visitors enter username and password. Username and password does not match to each other.
2. Visitors enter username and password. Username and password match to each other.

Python code for sign in is shown in figure 4.5. In first scenario, the program will reply a message "Username or password is invalid " as shown in figure 4.12. Visitors will be asked to enter their username and password again.

```
Do you have an account? [Yes/No]yes
Username:abc
Password:345
Username or password is invalid
Username:
```

Figure 4.12: Reply message for sign in scenario 1.

In second scenario, the program will reply a message "Logged in as : username " as shown in figure 4.13. Visitors are directed to chat room selection.

```
Username:abc
Password:123
Logged in as: abc
```

Figure 4.13: Reply message for sign in scenario 2.

4.4 User list synchronization

User list synchronization is a function that keeps updating list of username and password in the system. Users hence can access to chat rooms from different places with their registered account. There are 2 scenarios relating to user list synchronization as shown below.

1. User list will be updated when router is turned on.
2. After the first scenario finishes, user list will be updated for every specified period of time.

In first scenario, router will request for user list file from other routers to update its own user list. The process of requesting user list is identical to "Request and receive chat history messages". Instead of looking for a message contain room numbers, router will look for messages that contain "***@user", check for update and then update the user list file with **mergeuserfile()** function. Figure 4.14 to 4.16 show the python code of how this process work. As this process is similar to router failure handling in section 4.5, clear explanation is given in that section. Note that **mergeuserfile()** function does not sort user list by date as **mergefile()** function does to chat log files.

```
else:
    if myAddress not in address:
        if message == "ASK-FILE":
            print "SEND FILES"
            ff1 = open('log1.txt', 'r')
            ff2 = open('log2.txt', 'r')
            ff3 = open('log3.txt', 'r')
            ff4 = open('log4.txt', 'r')

            readuserlist = open('userlist.txt', 'r')

            file1 = ff1.read()
            file2 = ff2.read()
            file3 = ff3.read()
            file4 = ff4.read()

            alluser = readuserlist.read()
```

Figure 4.14: a part of python code for requesting user list file and chat logs file from neighboring routers.

```
elif "***@user" in message:
     container = message.split("***@user")
     mergeuserfile(container[1], "userlist.txt")
elif "***@userS" in message:
     print "Prepare Merge"
     container = message.split("***@userS")
     #print container[0]+container[1]
     message5.append(container[1])
     full_msg5 = "".join(message5)
     print "full_msg"+str(len(full_msg5))
     if len(full_msg5) >= message_size5:
             print "msg-size"+str(message_size5)
             mergefile(full_msg5, "userlist.txt")
     print "After Merge"
```

Figure 4.15: a part of python code in **main()** for requesting user list file and chat logs file from neighboring routers.

```
def mergeuserfile(file, filename):
        if "***@user" in file:
                container = file.split("***@user")
                file = container[1]

        masteruser = ""
        updatinguser = file.split("\n")
        myuserfile = open(filename, 'r')
        myuserfile = myuserfile.read()
        checkuserfile = myuserfile.split("\n")
        for index1 in range(len(checkuserfile)):
                if checkuserfile[index1]:
                        masteruser += checkuserfile[index1] + "\n"
        for index2 in range(len(updatinguser)):
                if updatinguser[index2] not in masteruser:
                        if updatinguser[index2]:
                                masteruser += updatinguser[index2] + "\n"

        writefile = open(filename,'w')
        writefile.write(masteruser)
        writefile.close()
```

Figure 4.16: a python code for userlist merging function.

In second scenario, a countdown timer-like function is executed. The timer function serves the system as an update keeper. In this project, the countdown timer-like function is **checktime()** and

is set at 5 minutes**.** Once the countdown finishes, it will execute another function , **timetoupdate()** , to open the user list file and send username and password to other router in the network by using **mergeuserfile()** function. Python codes for timer execution is shown in figure 4.17 and **checktime()** and **timetoupdate()** functions are shown in figure 4.18.

```
    roomNumber = raw_input("Enter Room Number [1,2,3,4]: ")
    if checkBug(roomNumber):
        isRoomSelected = 1
        f0 = open('log'+roomNumber +'.txt', 'r') #print chat history
        print(f0.read())
        f0.close()
else:
    message, address = s.recvfrom(8192)
    starttime = checktime(starttime)
```

Figure 4.17: a part of python code in **main()** that execute countdown timer

```
def checktime(starttime):
        start = starttime
        end = time.time()

        elapse = end - start

        if elapse > 5:
                timetoupdate()
                start = time.time()
                #print "user sent"
        return start

def timetoupdate():
        newuserlist = open('userlist.txt','r')
        chkuser = newuserlist.read()
        s.sendto("***@user"+chkuser,send_address)
```

Figure 4.18: **checktime()** and **timetoupdate()** functions in python language

4.5 Router failure handling

Request and receive chat history messages

Sometimes some routers may be down or fail during chat period that can cause messages that were sent not received by everyone. We implement the program to handle this kind of situation by given that every time the router turn on, the router will always call **sendRequestFiles()** function first to request the chat log files by sending the message "ASK-FILE" to other routers that are currently running on that time (We assume that there are some routers still running in that time, if not the chat log files cannot be updated until some or all routers are on)

After sending the "ASK-FILE" message, other routers that received this message will send the chat messages from every 4 chat room logs to the requested router (which is the router that just turn on), and also concatenate the chat messages with special character string "***@1Z" for indicate to the destination router that this is the message of chat contents, which the number before letter 'Z' is the number of log file (e.g. log1.txt). In case of the chat messages size is over MTU, we will concatenate the messages with another special character string "***@1SZ" to differentiate with the normal sending (messages is not over MTU), we will talk about this later in topic of "MTU Fragmentation" below.

During sending the chat history messages to other routers for merging and sorting, the parameter 'timeout' will start to run until it reach 'maxtime' (we set it to 1500) before go to chatroom selection state. We have to use timer in this case for prevent the requested chat history messages drop during sending, because the receive session (s.recvfrom()) do not open while the user is in select chat room state.

```
while True:
    try:
        if isRoomSelected == 0:
            if isSendAsk == 0:
                sendRequestFiles()
                isSendAsk =1
```

```
def sendRequestFiles():
    print "REQUEST FILES"
    s.sendto("ASK-FILE", send_address)
```

Figure 4.19: sendRequestFiles() is called in the main() function

```
if myAddress not in address:
    if message == "ASK-FILE":
       print "SEND FILES"
       ff1 = open('log1.txt', 'r')
       ff2 = open('log2.txt', 'r')
       ff3 = open('log3.txt', 'r')
       ff4 = open('log4.txt', 'r')

       readuserlist = open('userlist.txt', 'r')

       file1 = ff1.read()
       file2 = ff2.read()
       file3 = ff3.read()
       file4 = ff4.read()

       alluser = readuserlist.read()

       ##Handle MTU
       if len(file1) < MTU:
            print "file1 < MTU"
            s.sendto("***@1Z"+file1, send_address)
       else:
            print "file1 > MTU"
            if isSendHeader1:
                    header = "HEADER" + str(len(file1))
                    if (len(header) < 20):
                            blank = 20 - len(header)
                            header = header + (' ' * blank)
                    s.sendto("***1H"+header, send_address)
                    isSendHeader1 = False
            for data in spliter(file1,MTU):
                    print str(len(data))
                    s.sendto("***@1SZ"+data, send_address)
```

Figure 4.20: Implementation of how to send the requested chat messages

```
def main():
   maxtime         = 1000
   timeout         = 0
   isRoomSelected  = 0
   isSendAsk       = 0
   isFinishRequest = 0
   myAddress       = getip('wlan0')
```

```
while isFinishRequest == 0:
    if timeout == maxtime:
        isFinishRequest = 1
    else:
        timeout = timeout + 1

    message, address = s.recvfrom(8192)
    print "Merge File"
```

```
elif "***@1Z" in message:
    container = message.split("***@1Z")
    mergefile(container[1], "log1.txt")
elif "***@1SZ" in message:
    print "Merge fragment"
    container = message.split("***@1SZ")
    #print container[0]+container[1]
    message1.append(container[1])
    full_msg1 = "".join(message1)
    print "full_msg"+str(len(full_msg1))
    if len(full_msg1) >= message_size1:
            print "msg-size"+str(message_size1)
            mergefile(full_msg1, "log1.txt")
```

Figure 4.21: Implementation of how to receive the requested chat log messages

Merging messages

The received chat messages from each log files will enter the **mergefile(file,fileName)** function one by one. In the **mergefile(file,fileName)** function, it splits the content of chat messages that we have just requested line by line, and then compare with our local log file. If any chat messages is not in our local log file, we will merge that chat messages into our local log file. After finished merging, we have to sort the message by date&time before overwrite the updated messages in our local log file.

```
def mergefile(file, fileName):
    masterContent = ""
    someOneContent = file.split("\n")
    myfile = open(fileName, 'r')
    myfile = myfile.read()
    myContent = myfile.split("\n")
    for index1 in range(len(myContent)):
        if myContent[index1]:
                masterContent += myContent[index1] + "\n"
    for index2 in range(len(someOneContent)):
        if someOneContent[index2] not in masterContent:
                if someOneContent[index2]:
                        masterContent += someOneContent[index2] + "\n"

    sortBydate(masterContent,fileName)
    myfile.close()
```

Figure 4.22: Implementation of how to merge the requested chat messages

Sorting messages

The **sortBydate(file,fileName)** function handles sorting the merged messages to be according to date&time before overwrite them in our local log file. Since the messages that we record in the log file, including the date&time is in "string" type, so we have to split the date&time information from the whole messages and then convert it from "string" to "datetime" type first before performing date&time comparison. For comparison, we have implemented the loop to find the minimum date&time (oldest) first and then keep that line of messages (which includes username, IP Address, date&time, chat room no. and input messages) into the parameter "masterSorted". We keep doing this until we reach the maximum date&time (most recently).

After finished sorting all messages, the local log file will be opened to overwrite the updated chat messages. In this state all messages from every routers will be synchronization.

```
def sortBydate(file,fileName):
    masterSorted = ""
    splitLine  = file.split('\n')
    spliterSize = len(splitLine) - 1

    for outline in range(spliterSize):
        outdate = dt.strptime('Mon Jan 01 00:00:00 3000', "%a %b %d %H:%M:%S %Y")
        outString = ""
        for inline in range(spliterSize):
                inspliteLine = splitLine[inline].split('|')
                indateTemp   = inspliteLine[2]
                indate       = dt.strptime(indateTemp, "%a %b %d %H:%M:%S %Y")

                if splitLine[inline] not in masterSorted:
                        if outdate > indate :
                                outdate = indate
                                outString = splitLine[inline]

        masterSorted += outString + "\n"

    writefile = open(fileName, 'w')
    writefile.write(masterSorted)
    writefile.close()
```

Figure 4.23: Implementation of how to sort the requested chat messages

4.6 Data management control

Historical messages wiping

In this chat system we design to wipe our chat history everyday, so we open a thread for handling the history messages wiping, The **thread.start_new_thread(dispatch, ())** will be called via **callThread()** function since the beginning of **main()** when the router first running.
After the thread starting, **dispatch()** function will suspend the time by using "time.sleep(86400)" (24 hours = 86400 seconds) before wiping the chat history. After the suspension time, the **deleteHistory()** function will be called to wipe all chat history by overwrite all chat log files with blank("") string.

```
def callThread():
    #start thread dispatch
    thread.start_new_thread(dispatch, ())

def dispatch():
    time.sleep(86400) #delete history every 24 hours
    deleteHistory()

def deleteHistory():
    f1 = open('log1.txt', 'w')
    f1.write("")
    f1.close()

    f2 = open('log2.txt', 'w')
    f2.write("")
    f2.close()

    f3 = open('log3.txt', 'w')
    f3.write("")
    f3.close()

    f4 = open('log4.txt', 'w')
    f4.write("")
    f4.close()
```

Figure 4.24: Implementation of how to perform historical messages wiping

4.7 MTU (Maximum Transmission Unit) Fragmentation

Our router (wlan0) has an MTU of 1500 bytes, which is the maximum size of an IP packet that can be transmitted without fragmentation. In concept of MTU, if the UDP packets greater than the MTU size of the network, it will be automatically split up into multiple packets, and then reassembled by the recipient. If any of those multiple sub-packets gets dropped, then the receiver will drop the rest of them as well.
From the above information can imply that the packet loss rate of the larger packets will be higher than sending the smallest packets. So we try to decrease the packet loss rate by split the packet size to smaller fragment packets (not over MTU) first before sending to the destination.

In our program, if messages that want to send is over MTU (1500 bytes), the program will first send the total message sizes to the destination first before fragment the messages into smaller pieces by using **spliter(msg,n)** function, and then sending fragmented messages to the network.

```
#Split Message
def spliter(msg, n):
        for i in xrange(0, len(msg), n):
                yield msg[i:i+n]
```

```
##Handle MTU
if len(file1) < MTU:
     print "file1 < MTU"
     s.sendto("***@1Z"+file1, send_address)
else:
     print "file1 > MTU"
     if isSendHeader1:
             header = "HEADER" + str(len(file1))
             if (len(header) < 20):
                     blank = 20 - len(header)
                     header = header + (' ' * blank)
             s.sendto("***1H"+header, send_address)
             isSendHeader1 = False
     for data in spliter(file1,MTU):
             print str(len(data))
             s.sendto("***@1SZ"+data, send_address)
```

Figure 4.25: Implementation of how to handle MTU

In the receiver side, we will receive the total messages size that concatenate with the special character string “***1H”, which the number before letter ‘H’ indicate the number of log file (in this case is the size of log1.txt).

```
##MTU:before goto merge function, reassembly the fragmented messages first
if "***1H" in message:
     if message_header in message:
             message = message.split("***1H")
             data = message[1]
             message_size1 = int(str(data[len(message_header):]))
```

Figure 4.26: Implementation of how to receive the total messages size

After we know the whole size of the messages, all fragmented messages can be reassembly to the original larger messages. Our algorithm is that we collect/concatenate the fragmented messages into the parameter "full_msg1" continuously until its sizes greater or equal the original larger message size. After finishing reassembly process, we then sending the whole messages to **mergefile(file,fileName)** function to process further.

```
elif "***@1Z" in message:
     container = message.split("***@1Z")
     mergefile(container[1], "log1.txt")
elif "***@1SZ" in message:
     print "Merge fragment"
     container = message.split("***@1SZ")
     #print container[0]+container[1]
     message1.append(container[1])
     full_msg1 = "".join(message1)
     print "full_msg"+str(len(full_msg1))
     if len(full_msg1) >= message_size1:
             print "msg-size"+str(message_size1)
             mergefile(full_msg1, "log1.txt")
```

Figure 4.27: Implementation of how to reassemble the fragmented messages

Chapter 5
Conclusion and Future Work

Conclusion

Peer-to-peer computing is becoming more and more prevalent as people are using their own PCs to bypass central servers to connect directly with each other. This is changing the way people share information, collaborate, and how they learn. In conclusion our system is a multicasting chat system which is based on a text communication, it will be able to communicate with multiple users.

For implementing our Community Chat System (CCS) project, we found that there are several core problems that we have to accomplished as follows:
- Several rooms distribution among routers
- Distributed server (Multicast)
- Message merging and sorting
- MTU handling
- Error recovery for missing packets (Erasure coding codes)

As a matter of fact this project took us one month to be completed. First we had to understand the theoretical section behind each and every step and, where we have learned lots of things in networking and how it works .

Eventually, we could achieve and overcome every these core problems of this project, except only “Error recovery for missing packet (Erasure coding codes)” that we plan to improve it in our future work.

Future Work - Erasure code

We plan to apply Erasure coding. It is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces and stored across a set of different locations or storage media.

In mathematical terms, the protection offered by erasure coding can be represented in simple form by the following equation: n = k + m. The variable “k” is the original amount of data or symbols. The variable “m” stands for the extra or redundant symbols that are added to provide protection from failures. The variable “n” is the total number of symbols created after the erasure

coding process. For instance, in a 10 of 16 configuration, or EC 10/16, six extra symbols (m) would be added to the 10 base symbols (k). The 16 data fragments (n) would be spread across 16 drives, nodes or geographic locations. The original file could be reconstructed from 10 verified fragments.