SENT/UPLOADED 4/29 4:53 PM
DESIGN DOCUMENT

Project #3
Due Date: 4/13/2012

Group 2
Daeik Kim
Jonathan Child
Fred Knutson
Kevin Caton-Largent

Part I. Section I : NetProcess.java

NetProcess implements the network syscalls, accept() and connect(). These syscalls create a communication between two connection endpoints. They are called when you pass in a syscall which goes to handleSyscall, which in turns calls on handleConnect() or handleAccept(). handleConnect() connects a user program to a remote node using a host ID and a port number. Then the program will then call accept() to accept the connection. handleAccept() takes in the local port number for which the connections can be made on. NetProcess also has to be made to extend the UserProcess class the was created in project 2.

```
public class NetProcess extends UserProcess {


  public NetProcess() {
     super();
  }
  public int counter= 0;
  private static final int
     syscallConnect = 11,
     syscallAccept = 12;




  public int handleSyscall(int syscall, int a0, int a1, int a2, int a3) {


     switch (syscall) {
     case syscallConnect:
           return handleConnect(a0,a1);
     case syscallAccept:
           return handleAccept(a0);
     default:

        return super.handleSyscall(syscall, a0, a1, a2, a3);
           }
  }



  int handleConnect(int host,int port)
  {
    if (port < 0 || port >= portLimit) return -1;

    int fd = findOpening();
    if (fd!=-1)
    {



            Connection tempID = new Connection(...);

                NetSocket socket = new NetSocket(tempID);
                if (!socket.connect()) return -1;
                NetKernel.packetManager.openSockets.put(...);

                fileTable[fd]=socket;
                return fd;



    }
    else return -1;
}
int handleAccept(int port)
{

    int fd = findOpening();
    if (fd!=-1)
    {

            NetSocket socket = NetKernel.packetManager.waitingSockets[port].pollFirst();
            if (socket==null) return -1;
            socket.accept();
            NetKernel.packetManager.openSockets.put(socket.connection.getKey(), socket);
            this.fileTable[fd]=socket;
            return fd;

    }
    else return -1;
}
```

```
    LinkedList<Connection>  connections = new LinkedList<Connection>();
}
```

Part I. Section II : Connection.java

All of socket's information such as host port, host address, local port, and local address will be stored in a separate class called Connection. This is to simply our approach and attempt to put things in hierarchy so that we will come to use it for its specific tasks. Here's a possible implementation of the class.

```
public class Connection {

        int hostPort;
        int hostAddress;
        int localPort;
        int localAddress;
        public Connection()
        {
                hostPort = hostAddress = localPort = localAddress = -1;
        }
        public Connection (int HostAddress, int HostPort, int LocalPort,int LocalAddress)
        {
                localPort=LocalPort;
                hostAddress = HostAddress;
                hostPort = HostPort;
                localAddress = LocalAddress;
        }
        String getKey()
        {
                return "" + localAddress + "." + hostAddress + "." + localPort+ "." + hostPort;
        }


}
```

Part I. Section III : NTPacket.java

In order to make syscalls in network more efficient and more organized, it is necessary to create our own separate java class file which keep track of all packets sent and received. This class will be used in NetKernel instead of PostOffice, which was used before. These packets need to have lock, semaphore, and condition variables to control the flow. It will store linked list of packets and sockets along with hashmap of sockets to represent open sockets. These packets will be utilized for as data for our own socket.

- Firstly, the constructor class for NTPacket will create 3 threads for receiving, sending and appying timeouts to packets. Below is the possible implementations for implementing such tasks.

```
public NTPacket()
        {
                Runnable receiveHandler = new Runnable() {
                   public void run() { receiveInterrupt(); }
                };
                Runnable sendHandler = new Runnable() {
                   public void run() { sendInterrupt(); }
                };
                Machine.networkLink().setInterruptHandlers(receiveHandler,
                                                                        sendHandler);
                new KThread(new Runnable() {
                        public void run() { receivePackets(); }
                   }).fork();

                new KThread(new Runnable() {
                        public void run() { sendPackets(); }
                   }).fork();

                new KThread(new Runnable() {
                        public void run() { packetTimeouts(); }
                   }).fork();
        }
```

- receivePackets() involves looping through constantly and receiving the packet from networkLink() function inside Machine. It will make sure the packet is a valid packet with the length greater or equal to 2. Then it will save the contents of the packet into local variables to be utilized later. Then it will make sure the validity of the information of the packet and handle the packets accordingly. Below is the implementation of the function.

```
public void receivePackets()
        {

                while(true)
                {
                        //Call P() from a semaphore variable
                        Packet p = Machine.networkLink().receive();
```

```
                    //System.out.print("receiving:");
                    if(packet's content length is greater than 2)
                    {
                    int dstPort = p.contents[0];
                    int srcPort = p.contents[1];
                    //acquire the lock for the packet

                    if(packet's content length is greater than 3 and socket contains the key))
                        {
                            openSockets.get(getPacketKey(p)).handlePacket(p);
                        }
                    else if (packet's content length is greater than 3 and contents of packet is 1)
                        {
                            //Boolean value for connection pending is false
                            for(NetSocket n:waitingSockets[dstPort])
                            {
                                if(n.connection.hostAddress == p.srcLink
&&n.connection.hostPort == srcPort &&n.connection.localAddress == Machine.networkLink().getLinkAddress() &&
n.connection.localPort == dstPort){
                                //Boolean value for connection pending is true
                                    break;
                                }
                            }
                            if(!connectionAlreadyPending){
                      Connection tempID = new
Connection(p.srcLink,srcPort,dstPort,Machine.networkLink().getLinkAddress());
waitingSockets[dstPort].add(new NetSocket(tempID));
                                                                            }
                        }
                        else
                        {
                            packetList[dstPort].add(p);
                            packetSignal[dstPort].wakeAll();
                        }

                        packetListLock[dstPort].release();
                    }
                }
            }
```

- sendPackets() involves acquiring the lock and sending the packet by calling the send()

function. It will first make sure the size of the message Queue to see if it’s zero, and if it is it will put the condition variable for it to sleep. After that check, it will remove the item from the queue and send it via send() function. Below is the implementation for the function.

```
public void sendPackets()
	{
		sendPacketLock.acquire();
		while(true)
		{
			if (messageQueue.size() == 0)
				sendPacketSignal.sleep();
			Packet m = messageQueue.removeFirst();
			send(m);
		}
	}
```

- packetTimeouts() loops through constantly and utilizes the alarm class to perform timeouts. Then it will check to see Below is the possible implementation of the function.

```
public void packetTimeouts()
	 {
		 while(true)
		 {
			 NetKernel.alarm.waitUntil(20000);
			 for(Entry<String, NetSocket> e: openSockets.entrySet())
			 {
				 e.getValue().timeOutEvent();
			 }


		 }


	 }
```

- It is important to consider addition of the messages. It will require use of lock to make sure the full utilization of the resources. Then it will add the packet passed into the message queue and wake all the condition variable, then finally release the lock. Below is the implementation of the function addMessage.

```
public void addMessage(Packet m)
	{
		sendPacketLock.acquire();
```

```
        messageQueue.add(m);
        sendPacketSignal.wakeAll();
        sendPacketLock.release();

    }
```

- Nextly, the fully incorporate the packets and messages, it is essential that we have functions which gets message on port and wait for message on port. For getting message on port, it will acquire the lock then check for the size of the packet list in LinkedList and remove it to return it. On the other hand, the waitformessage function will require the lock and if the size is zero, then will put the condition to sleep. Then it will remove the packet from the list and return it.

```
public Packet getMessageOnPort(int port)
    {
        Packet p = null;
        packetListLock[port].acquire();
        if(packetList[port].size()> 0){
            p =packetList[port].remove();
        }
        packetListLock[port].release();
        return p;
    }

public Packet waitForMessageOnPort(int port)
    {


        Packet p = null;
        packetListLock[port].acquire();
        while(packetList[port].size() == 0){
            packetSignal[port].sleep();
        }
        p = packetList[port].remove();
        packetListLock[port].release();
        return p;
    }
```

Part I. Section IV: PostOffice.java

Minor change was made to ensure the delivery of the mail. It will loop constantly and send the message from the message queue. Below is the implementation of the change made.

```
private void deliverMail()
  {
          while(true)
          {
                  MailMessage m = messageQueue.pollFirst();
                  if (m!=null)
                  {
                          send(m);
                  }
                  KThread.yield();

          }

  }
```

Then it will create a thread in the constructor which implements runnable interface. Below is the implementation of the thread.

```
          KThread s = new KThread(new Runnable() {
                  public void run() { deliverMail(); }
            });
          s.fork();
```

Part I. Section IV: Socket.java

After the connection is established in NetProcess, it then calls on Socket.java.

connect() handles the CONNECTION action of the FSA. If socket state is closed, sends a SYN packet to the other side to do the handshake. It then changes the state to SYN_SENT. After sending the SYN_SENT it places the packet in the list of open sockets and then changes the state from SYN_SENT to ESTABLISHED.

```
          public boolean connect()
                  {
                          if(state == CLOSED)
                          {
                            Packet p=null
                            byte[] contents = new byte[8]
                            contents[0] = (byte)connection.hostPort
                            contents[1] = (byte)connection.localPort
                            contents[3] = SYN
```

```
                        try {
                                p = new Packet(connection.hostAddress, connection.localAddress, contents)
                                        this.sendPacket(p)
                                        state = SYN_SENT

// put packet onto list of open sockets
        NetKernel.packetManager.openSockets.put(connection.getKey, this)
                                        socketLock.acquire
                                        connectBlock.sleep
                                        socketLock.release
                                        state = ESTABLISHED
                                return true
                                } catch (MalformedPacketException) {
                                }
                        }
                        return false
                }
```

accept() function handles the accept of the FSA. if state is in SUM_RCVD then it queues packet with a new syn ack packet, and sets state to established.

```
        public boolean accept()
                {
                        state=ESTABLISHED
                        sendSYNACK
                        return true
                }
```

sendtSTP() sends out an STP packet.

```
        public void sendSTP()
                {
                        Packet p=null
                        byte[] contents = new byte[8]
                        contents[0] = (byte)connection.hostPort
                        contents[1] = (byte)connection.localPort
                        contents[3] = STP
```

```
		try {
			p = new Packet(connection.hostAddress, connection.localAddress, contents)
			this.sendPacket(p)
		} catch (MalformedPacketException) {
			System.out.println("malformed packet exeption sendSTP")
		}
	}
```

sendSYN() sends out an SYN packet.

```
public void sendSYN()
	{
		Packet p=null;
		byte[] contents = new byte[8]
		contents[0] = (byte)connection.hostPort
		contents[1] = (byte)connection.localPort
		contents[3] = SYN

		try {
			p = new Packet(connection.hostAddress, connection.localAddress, contents)
			this.sendPacket(p)
		} catch (MalformedPacketException) {
			System.out.println("malformed packet exeption sendSYN")
		}
	}
```

sendFIN() sends out an FIN packet.

```
public void sendFIN()
	{
		Packet p=null;
		byte[] contents = new byte[8]
		contents[0] = (byte)connection.hostPort
		contents[1] = (byte)connection.localPort
		contents[3] = FIN

		try {
			p = new Packet(connection.hostAddress, connection.localAddress, contents);
			this.sendPacket(p)
		} catch (MalformedPacketException) {
```

```
                System.out.println("malformed packet exeption sendFIN")
            }
        }
```

sendFINACK() sends out a FINACK packet.

```
public void sendFINACK()
        {
            Packet p=null;
            byte[] contents = new byte[8]
            contents[0] = (byte)connection.hostPort
            contents[1] = (byte)connection.localPort
            contents[3] = FINACK

            try {
                p = new Packet(connection.hostAddress, connection.localAddress, contents);
                this.sendPacket(p)
            } catch (MalformedPacketException) {
                System.out.println("malformed packet exeption sendFINACK")
            }
        }
```

sendSYNACK() sends out a SYNACK packet.

```
sendSYNACK()
        {
            Packet p=null
            byte[] contents = new byte[8]
            contents[0] = (byte)connection.hostPort
            contents[1] = (byte)connection.localPort
            contents[3] = SYNACK

            try {
                p = new Packet(connection.hostAddress, connection.localAddress, contents)
                NetKernel.packetManager.addMessage
            } catch (MalformedPacketException) {
                System.out.println("malformed packet exeption sendSYNACK")
            }
        }
```

close() function if the state is established sends out an STP packet and changes state to STOP_SENT.

```
close()
    {

        if(state == ESTABLISHED)
        {
            System.out.println("stop sent")
            sendSTP
            state = STOP_SENT
        }
    }
```

getNextPacket() returns the next packet from the received packets list.

```
getNextPacket()
    {
        socketLock.acquire
        for (go through list of received packets)
        {
            if (seqNoRead is within the list of received packets)
            {
                seqNoRead++
                Packet p = receivedPackets.remove(i)
                socketLock.release
                return p
            }

        }
        socketLock.release
        return null

    }
```

read() allows the connection to read from the network. Attempts to read a number of bytes from the socket. If it is closed and there are no bytes in the buffer, it will return -1, otherwise return the number of bytes read. It does not block.

```
public int read(byte[] buf,int offset,int length)
{
        int bytesRead = 0;
        if(offset + length <= buf.length)
        {

          if(Packet == null || PacketOff >= Packet.contents.length)
          {
            Packet = getNextPacket();
            PacketOff = 8;
          }
          while(Packet != null && bytesRead < length)
          {
                int amountToRead = Math.min(Packet.contents.length - PacketOff, length);
                amountToRead = Math.min(amountToRead, buf.length - bytesRead);

                System.arraycopy(Packet.contents, PacketOff, buf, offset + bytesRead, amountToRead);
                bytesRead += amountToRead;
                PacketOff += amountToRead;
                if(PacketOff >= Packet.contents.length){

                      Packet = getNextPacket();
                      PacketOff = 8;
                }
          }
        }
        if(bytesRead == 0 && state == CLOSED) return -1;
        return bytesRead;
      }
```

write() allows the connection to write to the network. Attempts to write a buffer of bytes to the socket. If the socket is not Established it returns -1.

```
public int write(byte[] buf,int offset,int length)
    {
          if(state == ESTABLISHED && offset + length <= buf.length)
          {
                Packet p=null;
                int bytePos = offset;
                int endPos = offset + length;
                while(bytePos < endPos)
                {
```

```
                int amountToSend = Math.min(Packet.maxContentsLength - 8, endPos - bytePos);
                byte[] contents = new byte[amountSend + 8];
                contents[0] = (byte)connection.hostPort;
                contents[1] = (byte)connection.localPort;
                System.arraycopy(buf, bytePos, contents, 8, amountSend);
                bytePos += amountToSend;
                setSeqNo(contents);
                try {

                    p = new Packet(connection.hostAddress, connection.localAddress, contents);
                    sendPacket(p);
                } catch (MalformedPacketException e) {
                        System.out.println("malformed packet exeption write()")
                }
            }
            return length;
        }
        return -1;
    }
```

handlePacket() handles the actions of the FSA as according to the protocol diagram. It sends the packets according to the state that the socket it is in.

```
handlePacket(Packet p)
        {
            seqNum
            if (p.contents == ACK)
            {
                    socketLock.acquire();
                    // checks if the incoming packet is the same as any of the unacknowledged
packets and updates the list of unacknowledged packets
                    for (loop through unacknowledgedPackets)
                    {
                            Packet temp = unacknowledgedPackets.get
                            if (temp == incoming packet P)
                            {
                                    unacknowledgedPackets.remove
                                    receivedAcks.add(seqNum)
                                    break;
```

```
                        }
                }
                socketLock.release
        }
        switch (p.contents)
        {
       // goes through every case and handles the corresponding action according to protocol
        case 0:
                if(seqNum >= lastSequenceNum) return
                switch(state)
                {
                case CLOSED:
                        break
                case SYN_SENT:
                        sendSYN
                        break
                case SYN_RCVD:
                        break
                case STOP_RCVD:
                case ESTABLISHED:
                        sendPacketAck(p)
                        socketLock.acquire
                        receivedPackets.add(p)
                        socketLock.release
                        break
                case STOP_SENT:
                        sendSTP
                        break
                case CLOSING:
                        sendFIN
                        break
                }

                break
        case SYN:

                switch(state)
                {
                case CLOSED:
                        state = SYN_RCVD
                        break
        case SYN_SENT:
            break
        case ESTABLISHED:
        case STOP_SENT:
        case CLOSING:
            sendSYNACK
            break
        }

        break
    case STP:
        switch(state)
        {
        case CLOSED:
            break
        case SYN_SENT:
            sendSYN
            break
        case SYN_RCVD:
            break
        case ESTABLISHED:
            state = STOP_RCVD
            lastSequenceNum = seqNum
            break
        case STOP_SENT:
            sendFIN
            state = CLOSING
            break
        case STOP_RCVD:
            break
        case CLOSING:
            sendFIN
            break
        }

        break;
    case SYNACK:
        socketLock.acquire
        connectBlock.wakeAll
        socketLock.release
        break
```

```
case ACK:
        switch(state)
        {
        case STOP_SENT:
                state = CLOSING
                sendFIN
                break
        }
        break
case FIN:
        switch(state)
        {
        case CLOSED:
                sendFINACK
                break
        case SYN_SENT:
                sendSYN
                break
        case SYN_RCVD:
                sendFINACK
                state = CLOSED
                break
        case ESTABLISHED:
                sendFINACK
                state = CLOSED
                break
        case STOP_SENT:
        case STOP_RCVD:
        case CLOSING:
                sendFINACK
                state = CLOSED
                break
        }
        break
case FINACK:
        switch(state)
        {
        case CLOSING:
                state = CLOSED
        }
        break
        }
    }
```

timeOutEvent() performs the TIMER action for various states in the FSA. If state is SYN_SENT, keep sending more SYN packets until a SYNACK signal is received. If state is established, then try to resend any unacknowledged packets. If state is closing then keep sending FIN packets until a FINACK signal is received.

```
timeOutEvent()
    {
        switch(state)
        {
        case SYN_SENT:
            sendSYN
            break
        case ESTABLISHED:
        case STOP_SENT:
            socketLock.acquire
            for(Packet p: unacknowledgedPackets)
            {
                NetKernel.packetManager.addMessage
            }
            socketLock.release
            break;
        case CLOSING:
            sendFIN
            break
        }

    }
```

sendPacket() sends out a packet.

```
sendPacket()
    {
        socketLock.acquire
        unacknowledgedPackets.add
        NetKernel.packetManager.addMessage
        socketLock.release
    }
```

sendPacketAck() sends out a packet acknowledgement.

```
public void sendPacketAck(Packet p)
{
        byte[] contents = new byte[8];
        contents[0] = (byte)hostPort;
        contents[1] = (byte)localPort;
        contents[3]= (byte) (p.contents[3] | ACK);
        System.arraycopy(p.contents, 4, contents, 4, 4);
        try {
            Packet packet = new Packet(hostAddress,localAddress,contents);
            NetKernel.packetManager.addMessage(packet);
        } catch (MalformedPacketException e) {
                    System.out.println("malformed packet exeption sendPacketAck")
          }
}
```

setSeqNo() sets the sequence number so that we know the order the packets get sent in.

```
public void setSeqNo(byte[] contents)
    {
        byte[] temp = new byte[4];
        bytesFromInt(temp, 0, 4, seqNo++);
        for (int i =0;i<4;i++)
            contents[4+i]=temp[i];
    }
```

Part II. Chat.c and Chatserver.c

Overall, the chat and chatserver are simple in concept. It just has to pass messages back and forth between users, and due to the simplicity of the system, it will only send messages to people when they are online, but to everyone that is online. As stated in the design, the connection port must be port 15. The different computers must be able to connect and disconnect at any time. The messages must be delivered to all chat clients. Order does not matter. The messages should not be truncated. The chat should be exited on “.”.

These functions, chat.c and chatserver.c, are basic in concept. chat.c will only obtain the message and send it to chatserver.c. Currently, no personal information is taken in or given out. If that should be added, it can be later.

chat.c

```
main {
    connect to server 15

    if can not connect to server
        error

    while (true) {
        read the line and store into writeBuffer with max length of 256 characters
        if the inputted character is "." disconnect
        sends writeBuffer

        while(there is something to print)
            "speak" message and clear buffer for new things to be inputted
    }
}
```

chatserver.c takes the message that was sent by chat and redistributes it to sendMessage, which will send the messages to the clients. It will also add people to the server by simply checking their port and give them the first free spot on the list.

```
chatserver.c
relayMessage {
    for i = 0 to number of clients
        while client[i] != null or sender
            send message and user identifier
}

main {
    while (1)
        handle connections and connections. check the port and put them into a spot.

        for i = 0 to number of clients
            if client[i] == null
                skip
            else
                creates the list of users
        for i = 0 to num of clients
            if client[i] != null
                read message
                relayMessage
}
```

Between my first draft and second draft, very little was actually changed as far as the functionality went. I realized that by having the disconnect where I had it would have disconnected everyone but the person who had typed it, as well as a few other small problems. The issues that I thought were going to be the hardest were actually fairly simple to process, especially numbering the users. I chose the easiest of the possible solutions, namely having a fixed number of users allowed to connect and allowing any number up until that to connect, the rest being unable to connect. If someone leaves, their status will be returned to -1 so their spot can be used the next time someone tries to access. Overall, there was very little to change between my original ideas and my final ones, I just wish that I had been less stupid when programming it. I tried to add a listen after we agreed that no listen statements were better.