The Specification of Project-2

EE122 Introduction to Communication Networks, Fall 2002
Ananth Rajagopala-Rao, Xuanming Dong, Khian Hao Lim

Version: Revision : 1.10

1 Overview

In Project 1, you developed a chat application using the client-server architecture. The goal of this project is to build a chat application with a user interface very similar to that of Project 1, but replacing the client-server model with a strictly peer-to-peer (P2P) model. In the P2P model, there is no separate server that provides services to the users of the system; the users cooperatively build the service that would otherwise have to provided by a dedicated server. While in the client-server model communication is only between a client and a server, in the P2P model every user is a peer and can communicate with other peers directly. In the P2P model, we don’t have to rely on a third-party to run the server and provide the service. But it does require all the users of the system to cooperate and trust each other.

The key difference between the P2P chat application and the client-server model is that now there is only one program, the chatpeer. This program does the job of both the chatclient (the user interface) and the chatserver (forwarding messages over the network to and from other users). To do this, we first build an Overlay Network on top of the IP network. An overlay network is a network at the application layer where each router (sometimes referred to as a node on the overlay) corresponds to an application program running on any computer connected to the Internet, and each overlay link (a virtual link) is itself an end-to-end path in the underlying IP network. To send a message from one user to the other, we must route on this overlay to the appropriate destination.

Because we must do routing at the application layer it makes it easier to think of the application itself as consisting of two separate layers (see Figure 1). The overlay routing layer sits directly on top of the transport layer and moves data from any node\(^1\) in the overlay to any other node. The the messaging layer uses the service provided by the overlay routing layer to build the instant messaging service for the end-user.

\(^1\)the terms node and peer are used interchangeably, and refer to a single instance of the chatpeer program on the overlay

![Diagram of Chatpeer application layers](image)

**Figure 1.** The chatpeer application comprises two layers
1.1 The Overlay Routing Layer

**Service:** Construct, maintain and route on an overlay network. Packet forwarding in this layer is best effort, and reliability must be implemented at the higher layer. Packets are not acknowledged or retransmitted hop-by-hop on the overlay.

**Interface:** Send a packet (best-effort) to any destination on the overlay network.

**Protocol:** The protocol uses UDP to exchange both control packets (to maintain and construct the overlay) and data packets. Each node on the overlay is associated with a unique 32-bit identifier (NodeId) which is used for addressing on the overlay (just as IP addresses are used for routing in the Internet).

Unlike a routing protocol at the network layer, the overlay routing protocol has complete freedom over the topology it has to route over. In other words, since overlay links are actually virtual links made of end-to-end paths in the Internet, they can be added to the topology or torn down at will. Furthermore, we can add a link between any two nodes in the overlay, we are not constrained by any physical topology.

For the sake of this project, we have chosen a very simple ring topology. Every node is “connected” by a virtual link to exactly two other nodes to form a ring, and the nodes appear on the ring in the order of their NodeIds. If we think NodeIds as coming from a circular identifier space where 0 follows \(2^{32} - 1\), every node forms links only to nodes that are closest to itself on either side in this circular identifier space. Figure 2 illustrates the circular identifier space and the virtual topology formed by the Nodes in the system. To explain the concept, we have shown a 6-bit identifier space (0...63) instead of the full 32-bit identifier space. This 6-bit identifier space is used in later examples also.

1.2 The Messaging Layer

**Service:** Allow a user to “join” the overlay network with a username of his choice. Once in the overlay, the user can send a message reliably to any other user on the overlay (knowing only intended receivers’ username). The send fails if the receiver is not online (or if no receiver with the given username exists). Multicast (sending the same message to more than one user) and listing of all current users is not supported.

**Interface:** The user interface is very similar to that of Project 1, but for the fact that certain commands from Project 1 are not supported. The exact details of the user interface are given in Section 4.
Protocol: Firstly, we need a method to find the NodeId given the username a user wants to send a message to, since addressing in the overlay routing layer is in terms on the NodeId, not the username. For this, we assume the existence of a well-known hash function (specified as part of the protocol) to map usernames to 32-bit identifiers.

Secondly, this layer has to implement acknowledgements, timeouts and retransmissions. Unlike Project 1, you are required to implement fine-grained timers (explained later) for this project.

2 The chatpeer program

Usage: chatpeer <username> <port> [<peer_address> <peer_port>]

chatpeer is a single threaded program that is run by every user of the chat application. Once started, the chatpeer creates a UDP socket at the given <port>, which it uses to send and receive messages. <peer_address> and <peer_port> is the address and port of any other chatpeer program that is already running (used by the new user to bootstrap and join the ring). These are not specified for the first user in the system, and hence these two parameters are optional.

The files chat.h and chat.cc have been provided for your convenience. The packet types and a few other parameters are defined in chat.h. The code for some useful functions is given in chat.cc. A starter Makefile is also provided.

2.1 Structure of the chatpeer program

Note that the chatpeer program implements both the messaging layer and the overlay routing layer. Thus, the boundary between the messaging layer and the overlay routing layer is inside a single program, and the only reason we think of the application as comprising two layers is for the sake of modularity. In other words, the concept of having two separate layers is an effective way to organize and test your code. The interface between the two layers will consist of regular function calls. Every time the messaging layer has a packet to send out, it calls a function for sending data defined in overlay routing layer. Similarly, every time the overlay routing layer receives a packet that has to be delivered to the messaging layer, it calls a message handler function defined in the messaging layer.

Having a clean separation between the two layers in your code will make it easier to read and debug. Of course, some data will have to be shared between both layers, but try to keep this at the minimum possible.

3 Timers

You will have to use timers for correctly implementing retransmissions of lost packets. This is required both in the messaging layer (for lost messages) and the overlay routing layer (for ring maintenance). For example, whenever the messaging layer sends a packet, a count-down timer is set to expire MLP_TIMEOUT milliseconds later.

It was mentioned earlier that you must use fine-grained timers. This means the action corresponding to a timer (like retransmission of a packet from the messaging layer) must be performed as soon as the timer expires. This means that you must use the timeout argument\(^3\) in the select function call.

Also, you may need to keep track of multiple timers at the same time, if for example the user types a second message before the acknowledgement for the first message she typed has arrived. The timeout argument to select will be dictated by the timer that is going to expire the sooner. For this reason, you may find it useful to maintain a priority queue ordered by expiration time to keep track of all outstanding timeouts. This queue will have to be shared by both layers since both layers use timers.

\(^2\)This function call is sometimes referred to as an up-call.

\(^3\)Look at the last argument (struct timeval *timeout) in the man page for select (2).
4 User Interface

User input (which we call UserInput) is obtained from standard input. Each UserInput fits into a single line and the chatpeer processes the UserInput when the user types <Enter> key. If user input does not conform to any of the following, chatpeer prints the following to stdout and the non-conforming UserInput is ignored.

incorrect userInput format

The possible UserInputs are:

1. UserInput_Check
   - Function: Check if somebody with the given username is online (connected to the overlay)
   - Format: /check username

2. UserInput_Message
   - Function: Send a message from this chatpeer to another chatpeer connected to the overlay using the username specified in this UserInput.
   - Format: /mesg username message

3. UserInput_Help
   - Function: List all the commands possible for this chatpeer. The chatpeer prints a list of possible UserInputs to stdout.
   - Format: /help
     You can use any format to print the list of commands to stdout. This command is for the sake of completeness of the chatpeer.

4. UserInput_Quit
   - Function: chatpeer prints the following to stdout and terminates after closing its socket.
     quitting chatpeer gracefully
   - Format: /quit

5 The Messaging Layer Protocol

5.1 Packet Format

The packet format for the Messaging Layer Protocol is shown in Figure 3.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Sequence Number</th>
<th>SLen</th>
<th>SrcUserName</th>
<th>DLen</th>
<th>DstUserName</th>
<th>Message</th>
</tr>
</thead>
</table>

Figure 3. The Messaging Layer Protocol Packet Format

Type: The packet type can be one of MLP_CHECK, MLP_MESSAGE, MLP_ACK or MLP_NACK. For message types MLP_CHECK and MLP_ACK, the Message part of the packet is empty (doesn’t exist). The MLP_CHECK packet is used to check if a given user is online. The MLP_MESSAGE packet is used to send a message to a given username. The MLP_ACK is used to acknowledge both MLP_MESSAGE and MLP_CHECK by the receiver. The MLP_NACK packet is generated when the specified username in an MLP_CHECK or an MLP_MESSAGE packet does not exist.
Sequence Number: A 4-byte sequence number is included in every MLP_MESSAGE and MLP_CHECK packet by the sender. Exactly how the sequence number is chosen is not specified in the protocol; the only requirement is that different MLP_MESSAGE and MLP_CHECK packets from the same sender have unique sequence numbers. When the receiver sends the MLP_ACK packet, the sequence number from the message which is being acknowledged is copied into this field in the packet.

SrcUserName, SLen: The username of the sender, and the length of this username respectively.

DstUserName, DLen: The username of the intended receiver, and the length of this username respectively.

5.2 Data Structures

Every node must maintain the following data structures to keep track of acknowledgements, retransmissions and duplicates.

- A suitable data structure to keep track of unacknowledged packets, the number of retransmissions (initialized to zero and incremented every time the packet is retransmitted), and the timeouts.
- A queue of the sequence numbers of the MLP_MESSAGE packets received in the recent past. The reason this is required is that if the acknowledgement for the message is lost, the sender will retransmit the MLP_MESSAGE packet, but the receiver should not display the same message twice at the terminal. Entries in this queue expire after time MLP_MAX_DELAY, where MLP_MAX_DELAY is some reasonable upper bound on the maximum delay a packet can experience in the network (will be defined in a header file given to you).

The pseudo-code for the message handler in the messaging layer is given below

HandleMLPPacket(p)
switch (p.type)
case MLP_MESSAGE:
    if (p.DstUserName ≠ myUserName)
        sendNACK(p)
    else if (!checkIfMessageIsAlreadySeen(p.SequenceNumber, p.SrcUserName))
        print the incoming message to stdout
        addSeenMessage(p.SequenceNumber, p.SrcUserName)
        sendACK(p)
case MLP_CHECK:
    if (p.DstUserName ≠ myUserName)
        sendNACK(p)
    else
        sendACK(p)
case MLP_ACK:
    timer = findTimer(p.SequenceNumber)
    print a message telling that the user is online
    delete(timer)
case MLP_NACK:
    timer = findTimer(p.SequenceNumber)
    print a message telling that the user isn’t online
    delete(timer)

5.3 The fine print

All two and four byte numeric fields (type, lengths and sequence numbers) in the packets are in network byte order. The Length field in the packet is the total length of the packet including the header. The hash function
<table>
<thead>
<tr>
<th>Constant</th>
<th>What it represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXNAMELEN</td>
<td>The maximum length of a valid user name</td>
</tr>
<tr>
<td>MLP_TIMEOUT</td>
<td>The timeout (in milliseconds) for retransmission of a packet</td>
</tr>
<tr>
<td>MLP_MAX_RETRIES</td>
<td>The number of times a packet must be retransmitted before giving up</td>
</tr>
<tr>
<td>MLP_MAX_DELAY</td>
<td>The time for which you should keep track of a message already printed to the terminal</td>
</tr>
</tbody>
</table>

Table 1. Constants defined in chat.h for the messaging layer

![SuccessorNode diagram](image)

Figure 4. Examples of how the SuccessorNode function is defined

(get_user_name_hash) to map usernames to 32-bit identifiers is defined\(^4\) in chat.cc. Table 1 shows the constants defined in chat.h and what they represent. Table 2 shows output to stdout that your program must generate. Please call fflush(stdout) after every call to printf or cout.

### 6 The Overlay Routing Protocol

As stated earlier, the overlay topology is a ring in which nodes are arranged in order of their NodeIds. Think of every NodeId as being chosen from a circular identifier space where \( x + 1 \) follows \( x \) for \( 0 \leq x < 2^{32} - 1 \) and 0 follows \( 2^{32} - 1 \). We now define SuccessorNode(\( x \)) as the node with NodeId=\( x \) if such a node exists, or the node whose NodeId immediately follows \( x \) on the circle otherwise (see Figure 4 for examples).

The job of the Overlay Routing Protocol is to route a packet to SuccessorNode(\( x \)) for any \( x \), starting from any node on the overlay. Thus, to send a packet to a given username, we have to address it to get_user_name_hash(username) on the overlay.

### 6.1 Forwarding Algorithm

For forwarding, every node N in the system is required to keep track of the NodeId, the address (the IP address) and the port of its successor node (successorNode) and the NodeId of its predecessor (predecessorNode) node on the ring. Data packets flow only in the clockwise direction along the ring. Figure 5 shows how the two layers work together to transport messages and their acknowledgements. Node N0 sends an MLP_MESSAGE packet to node N1. This packet is delivered at N1 by the overlay routing protocol using N10 as an intermediate hop. On receiving this packet the messaging layer at N1 generates an MLP_ACK packet that gets routed by the routing layer to N0 through N20, N30 and N33.

---

\(^4\)The hash function is the first 4 bytes of the MD5 checksum of the username taken in network byte order. If you are using C or C++, you don’t have to worry about this, you can use the code already provided to you.
### Table 2. Messages to be printed to stdout. Note that some messages appear in two lines in the “output” column simply because they don’t fit in the same line, you should not print a newline in the middle.

<table>
<thead>
<tr>
<th>Event</th>
<th>Output to stdout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message received for the user (message is not a duplicate)</td>
<td>&lt;SrcUserName&gt;: &lt;Message&gt;</td>
</tr>
<tr>
<td>Duplicate message received for the user</td>
<td>MLP: Received duplicate message from &lt;UserName&gt;</td>
</tr>
<tr>
<td>Received an MLP_CHECK packet with correct username</td>
<td>MLP: Received check, sending ACK to &lt;SrcUserName&gt;</td>
</tr>
<tr>
<td>Received an MLP_CHECK packet with incorrect username</td>
<td>MLP: Received check for &lt;DstUserName&gt;, sending NACK to &lt;SrcUserName&gt;</td>
</tr>
<tr>
<td>Received an MLP_MESSAGE packet with incorrect username</td>
<td>MLP: Received message for &lt;DstUserName&gt;, sending NACK to &lt;SrcUserName&gt;</td>
</tr>
<tr>
<td>Received ACK for a /check command</td>
<td>&lt;UserName&gt; is online</td>
</tr>
<tr>
<td>Received NACK for a /check command</td>
<td>&lt;UserName&gt; is not online</td>
</tr>
<tr>
<td>Received ACK for a /msg command</td>
<td>MLP: Received ACK for message from &lt;UserName&gt;</td>
</tr>
<tr>
<td>Received NACK for a /msg command</td>
<td>Could not send message, &lt;UserName&gt; is not online</td>
</tr>
<tr>
<td>Timeout for an MLP_CHECK or an MLP_MESSAGE packet which leads to a retransmission</td>
<td>MLP: Timeout, retransmitting &lt;packetType&gt; packet to &lt;UserName&gt; for the i-th time</td>
</tr>
<tr>
<td>Timeout for an MLP_CHECK or an MLP_MESSAGE packet, but the maximum number of retransmissions is reached</td>
<td>MLP: Timeout, maximum number of retransmissions reached for &lt;packetType&gt; packet to &lt;UserName&gt;</td>
</tr>
<tr>
<td>Any unexpected packet (type unknown, wrong length etc.)</td>
<td>MLP: Unexpected packet received.</td>
</tr>
</tbody>
</table>
N0 sends a message to N1 and N1 sends an ACK to N0
(all packets flow in the clockwise direction)

Figure 5. The interaction between the two layers in routing messages along the ring

The pseudo-code for the forwarding algorithm is given below. The function WrapBetween(l,h,x) checks if \( x \) lies between \( l \) and \( h \) in the circular identifier space.

\[
\text{WrapBetween}(l,h,x) \\
\text{if } (l < h) \\
\quad \text{return } ((x > l) \text{ and } (x < h)) \\
\text{else} \\
\quad \text{return } ((x > l) \text{ or } (x < h))
\]

\[
\text{Forward(packet, destinationId)} \\
\text{if } ((\text{destinationId} = \text{NodeId}) \text{ or } \text{WrapBetween(predecessorNode.NodeId, NodeId, destinationId)}) \\
\quad \text{stripORLHeader(packet)} \\
\quad \text{HandleMLPPacket(packet)} \\
\text{else} \\
\quad \text{send(packet, successorNode.address, successorNode.port)}
\]

For the forwarding algorithm to work every node needs to know about its successor. But if the successorNode fails, the ring is broken and forwarding would fail. Therefore, to be able to recover from the failure of up to \( k \) nodes in the system, each node has to maintain a list of \( k + 1 \) successors.

To sum up, each node has to maintain data structures that look like this.

```
struct Peer 
  U32 NodeId 
  U32 address 
  U16 port 
Peer predecessorNode 
Peer successorNodes[k+1]
```
Please do not confuse the successorNodes array with successorNode (the first successor of a given node on the ring, same as successorNodes[0]) or SuccessorNode(x) (defined earlier).

### 6.2 Packet Formats

#### 6.2.1 Data Packet

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>InitAddr</th>
<th>InitPort</th>
<th>InitId</th>
<th>DestinationId</th>
<th>Data from the Messaging Layer</th>
</tr>
</thead>
</table>

**Figure 6. The Overlay Routing Layer Header Format.**

**Type:** This is set to ORL\_DATA for a data packet.

**Length:** The total length of the packet.

**InitAddr:** This is the address of the initiator of the packet on the overlay. This field (along with InitPort and InitId) is filled in only the first time packet is received from the messaging layer for forwarding. It does not get over-written at intermediate hops.

**InitPort:** The port of the initiator of the packet.

**InitId:** The NodeId of the initiator of the packet.

**DestinationId:** The packet has to be routed to SuccessorNode(DestinationId).

#### 6.2.2 Control Packets

There are three types of control packets for the overlay routing layer.

1. The stabilize packet (Type = ORL\_STABILIZE).
2. The notify packet (Type = ORL\_NOTIFY).
3. The join request packet (Type = ORL\_JOIN).

The stabilize and the notify packets are similar to the KEEP\_ALIVE messages used in project 1. Every node sends the stabilize packet to its successorNode every STABILIZE\_PERIOD. On receiving a stabilize packet, a node sends a notify packet back to the sender of the stabilize packet. The join request packet is used by a new node to join into the ring at startup.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>FromId</th>
</tr>
</thead>
</table>

**Figure 7. The Stabilize Packet Format**

The list of up to k successors in the notify packet is the first k elements of the successorNodes array at the sender. This list can be shorter than k when the ring is itself smaller than k + 1 nodes, or when recovering from failures.

The pseudo-code for the Stabilize-Notify protocol for ring maintenance is given below. The Stabilize() function is called by every node every STABILIZE\_PERIOD.

Version: Revision : 1.10
Stabilize()
  send(stabilizePacket, successorNodes[0].address, successorNodes[0].port)
  receive(notifyPacket)
  if (timeout)
    if (maximum number of retries reached)
      shift successorNodes array left by one
      reset retransmission counter and send a stabilize packet to the new successor
      return
    else
      retry sending the stabilizePacket
  HandleNotify(notifyPacket)

HandleNotify(notifyPacket)
  if (NodeId = notifyPacket.FromId)
    // Same username or collision in the hash function
    print a message and exit
  if (WrapBetween(NodeId, successorNodes[0].NodeId, notifyPacket.PredecessorNode))
    // A new node just joined after me
    successorNodes[0] = notifyPacket.PredecessorNode
    successorNodes[1].NodeId = notifyPacket.FromId
    successorNodes[1].address = source address of notifyPacket
    successorNodes[1].port = source port of notifyPacket
    successorNodes[2..k] = notifyPacket.SuccessorList[0..(k-2)]
  else
    successorNodes[0].NodeId = notifyPacket.FromId
    successorNodes[0].address = source address of notifyPacket
    successorNodes[0].port = source port of notifyPacket
    successorNodes[1..k] = notifyPacket.SuccessorList[0..(k-1)]

HandleStabilize(stabilizePacket)
  // If our old predecessor is dead or if a new node joined between me and my old predecessor,
  // then update my predecessor information.
  if (((getCurrentTime() - lastPredecessorStabilizeSeenTime)
    > (STABILIZE_PERIOD + STABILIZE_TIMEOUT * (STABILIZE_MAX_RETRIES+1)) or
    (WrapBetween(predecessorNode.NodeId, NodeId, stabilizePacket.FromId)))
  predecessorNode.NodeId = stabilizePacket.FromId
  predecessorNode.address = source address of stabilizePacket
  predecessorNode.port = source port of stabilizePacket
  else if (stabilizePacket.FromId = predecessorNode.NodeId)
    // Just an update from our current predecessor
    lastPredecessorStabilizeSeenTime = getCurrentTime()
send(notifyPacket, source address of stabilizePacket, source port of stabilizePacket)

The variable “lastPredecessorStabilizeSeenTime” is used to keep track of the last time at which a stabilize packet was received form the predecessorNode. If more than (STABILIZE_PERIOD + STABILIZE_TIMEOUT \times (STABILIZE_MAX_RETRIES+1)) has elapsed since the last time at which a stabilize packet was received from the predecessorNode, it means that the predecessor has failed, and needs to be updated.

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>NewAddr</th>
<th>NewPort</th>
<th>NewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 9. The Join Request Packet Format

The join request is sent by a new node to the port and address of a peer obtained through the command line. Once on the overlay, the join request is *routed just like a data packet* to the SuccessorNode(NewId). This node then sends a notify packet back to the new node (whose address and port are available in the join request packet).

Once the notify packet from SuccessorNode(NewId) reaches the new node, it gets a list of successors and is good to go. The next time it sends a stabilizePacket to its successor, the predecessorNode at the successor is updated and the new node is part of the ring.

HandleJoin(joinPacket)
  if (WrapBetween(predecessorNode.NodeId, NodeId, joinPacket.NewId) or (joinPacket.NewId = NodeId))
  else
    send(joinPacket, successorNode.address, successorNode.port)

Join() //called by a node when it is starting up
send(joinPacket, peerAddress, peerPort) // peerAddress and peerPort from command line
receive(notifyPacket)
  if (timeout)
    if (maximum number of retries reached)
      print a message and exit
    else
      retry sending the joinPacket
  HandleNotify(notifyPacket)

HandleNotify(notifyPacket)

Figure 10 shows the various steps by which a new node joins the ring. The steps are labelled in circles in the order in which they occur.

1. N40 sends a join packet to N20 (whose address and port are given through the command line). This packet gets routed from N20 to SuccessorNode(45)=N0.
2. N0 responds with a notifyPacket to N40. The successorNodes array at N0 is now up to date.
3. N40 sends a stabilizePacket to N0. When this packet is received at N0, the predecessorNode is updated.
4. N0 sends a notifyPacket to N40.
5. When the STABILIZE_PERIOD expires at N33, it sends a stabilizePacket to N0.
6. N0 sends a notifyPacket to N33 saying its predecessor is N40.
7. N33 updates its successor to N40, and now N40 is part of the ring.
The ring before N40 joins

N40 sends out the Join Packet
N0 sends back the Notify Packet to N40

Now N40 is in the ring

Figure 10. The various steps by which a new node joins the ring
<table>
<thead>
<tr>
<th>Constant</th>
<th>What it represents</th>
</tr>
</thead>
<tbody>
<tr>
<td>STABILIZE_PERIOD</td>
<td>The period (in milliseconds) between two stabilize packets sent to the successor</td>
</tr>
<tr>
<td>STABILIZE_TIMEOUT</td>
<td>The timeout (in milliseconds) for retransmission of a stabilize packet</td>
</tr>
<tr>
<td>STABILIZE_MAX_RETRIES</td>
<td>The number of times a stabilize packet must be retransmitted before giving up</td>
</tr>
<tr>
<td>JOIN_TIMEOUT</td>
<td>The timeout (in milliseconds) for retransmission of the join request packet</td>
</tr>
<tr>
<td>JOIN_MAX_RETRIES</td>
<td>The number of times a join request packet must be retransmitted before giving up</td>
</tr>
<tr>
<td>NUM_FAILURES</td>
<td>The number of simultaneous failures the system must tolerate. This is same as k used in the discussion.</td>
</tr>
</tbody>
</table>

Table 3. Constants defined in chat.h for the routing layer

6.3 The fine print

Table 3 shows the constants defined in chat.h along with what they represent. Table 4 shows what should be printed out to stdout and when it should be printed out. Again, you must call fflush(stdout) after every call to printf or cout.

Finally, a word of caution about the pseudo-code provided. The pseudo-code has been given only to convey at a high level what needs be done. Translating the pseudo-code line-by-line into C or C++ will most likely not work. In particular, a lot of checking for corner cases has been omitted. The Join() and Stabilize() algorithms are given in blocking input/output style; you have to implement them using only a single thread and non-blocking input/output.

7 Checkpoint

For the checkpoint (due in three weeks), you have to implement the messaging layer (including timeouts) and the user interface fully. In the overlay routing layer, you have to implement the forwarding algorithm only (Stabilize, Notify and Join need not be implemented). NUM_FAILURES will be set to zero so that each node has to maintain only one successorNode. This successorNode (address, port and NodeId) and the predecessorNode (NodeId only\(^5\)) will be provided through the command line. Thus, the invocation of the chatpeer program (for the purposes of the checkpoint only) will be

Usage: chatpeer <username> <port> <successor address> <successor port> <successor id> <predecessor id>

8 For the adventurous and the brave

Once you have implemented the chatpeer program with the specifications given above, you can try adding more functionality to it for bonus points. A few ideas for enhancing the functionality of the chatpeer are listed here.

- Try to optimize the number of hops taken by messages on the overlay. Can you do the optimizations in such a way that your program is still compatible (must work) with other nodes in the system that do not implement the same optimization? There are a number of ways to do this.
  - Try and make use of the other entries in the successorNodes array while forwarding packets
  - “Snoop” on the InitAddr, InitPort and InitId of incoming packets and maintain a table. Use entries from this table while forwarding. How do you deal with failures when you do this?

\(^5\)The forwarding algorithm needs only the NodeId of the predecessor, the address and port are required only for the notify packet.
<table>
<thead>
<tr>
<th>Event</th>
<th>Output to stdout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending a stabilize packet (including sends after a timeout)</td>
<td>ORL: Sending ORL_STABILIZE to &lt;NodeId&gt;</td>
</tr>
<tr>
<td>Sending a notify packet</td>
<td>ORL: Sending ORL_NOTIFY to &lt;NodeId&gt;</td>
</tr>
<tr>
<td>successorNodes[0] changed because of a new node in the system</td>
<td>ORL: New Successor &lt;NodeId&gt; joined</td>
</tr>
<tr>
<td>(detected in the second if condition of HandleNotify)</td>
<td></td>
</tr>
<tr>
<td>successorNodes[0] changed because the current successor failed</td>
<td>ORL: Successor failed. New Successor is &lt;NodeId&gt;</td>
</tr>
<tr>
<td>(detected when maximum number of retransmissions is reached for</td>
<td></td>
</tr>
<tr>
<td>the stabilize packet)</td>
<td></td>
</tr>
<tr>
<td>predecessorNode changed because a new node joined the system</td>
<td>ORL: New Predecessor &lt;NodeId&gt; joined</td>
</tr>
<tr>
<td>(detected in the second if condition of HandleStabilize)</td>
<td></td>
</tr>
<tr>
<td>predecessorNode changed because the current predecessor failed</td>
<td>ORL: Predecessor failed. New Predecessor is &lt;NodeId&gt;</td>
</tr>
<tr>
<td>(detected in the first if condition of HandleStabilize)</td>
<td></td>
</tr>
<tr>
<td>Sending a join request packet (including retransmissions)</td>
<td>ORL: Sending ORL_JOIN</td>
</tr>
<tr>
<td>Joining the ring failed (no reply to the join request)</td>
<td>ORL: Join failed</td>
</tr>
<tr>
<td>Collision in NodeIds detected</td>
<td>ORL: NodeId being used by somebody else</td>
</tr>
<tr>
<td>successorNodes array is empty when trying to shift left</td>
<td>ORL: Out of successors</td>
</tr>
<tr>
<td>Unexpected packet received</td>
<td>ORL: Unexpected packet received</td>
</tr>
</tbody>
</table>

**Table 4. Messages to be printed to stdout**

• Try and implement efficient multicast (addressing the same message to multiple users) in this system. One possible idea is to have sorted list of destination NodeIds in the overlay routing layer header. What do you do about acknowledgements?

Even if you are shooting for bonus points, you must submit a vanilla version of chatpeer without any of the enhancements. You should then submit the enhanced version along with a report, saying what you did and why you think it is a good idea (instructions will be provided later at the course web-site).