CSS 434
Program 1: Two Chat Systems

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

This assignment learns how to use Java-based TCP communication through a design of chat client and server program. We also implement consistent ordering in broadcasting messages from one client to all the others through a central server as well as causal ordering in exchanging messages among clines without using central server.

2. Message Ordering

As we study in the class, we need to distinguish several message ordering schemes depending on distributed applications. These include absolute ordering, consistent ordering, causal ordering, and first-in-first-out ordering (listed here from the tightest to the loosest consistency). Although absolute ordering is accurate to keep track of who spoke first and who responded second in Internet chatting, it needs a global clock among clients, which is almost impossible over Internet. Therefore, we implement consistent ordering and causal ordering.

In consistent ordering, messages are simply delivered in the same order to each client regardless of their stamp. This means, although message A was created before message B, message B may be delivered to all chat participants first, but it is guaranteed that all users receive message B and thereafter message A. We use a centralized algorithm to implement this consistent ordering. More specifically, each chat client needs to contact a central chat server that then broadcasts the client message to all the clients, in which manner the server works as a focal point of consistent message ordering.

In causal ordering, a message is delivered to each receiver before it receives all the subsequent messages generated by the third party. For this purpose, each computing node maintains a vector of stamps, each counting send/receive events at node *i*. (where 0 <= *i* < #hosts). When a sender sends a message, it also includes its own vector stamp in the message. The receiver receives the message only when all stamps in the source *j*'s vector are smaller or equal to those in the receiver's vector except stamp j in the source j's vector is one larger than that in the receiver's vector.

Part I: Chats in Consist Order

A. Chat Client

A chat client program should be invoked with arguments such as a client name, a server IP name, and a server port:

```
[css434@uw1-320-20 hw1]$ java ChatClient mickey uw1-320-10 12345
```

The client program establishes a socket connection to the server, generates a parit of data input and output streams from this connection, sends its client name with writeUTF( ), and finally goes in an infinitive loop where it repeats reading a new keyboard input, (i.e., stdin) if there is any, writing this message to the server, checking if there is a new message arrived from the server, and displaying it to the display, (i.e., to stdout). If the client detects EOF through stdin, it leaves this loop and terminates itself. The following shows the client program, (ChatClient.java):

```
/**
 * ChatClient.java:<p>
 * realizes communication with other clients through a central chat server.
 *
 * @author  Munehiro Fukuda (CSS, University of Washington, Bothell)
 * @since   1/23/05
 * @version 2/5/05
 */
import java.net.*;          // for Socket
import java.io.*;           // for IOException

public class ChatClient {
    private Socket socket;           // a socket connection to a chat server
    private InputStream rawIn;       // an input stream from the server
    private DataInputStream in;      // a filtered input stream from the server
    private DataOutputStream out;    // a filtered output stream to the server
    private BufferedReader stdin;    // the standart input

    /**
     * Creates a socket, contacts to the server with a given server ip name
     * and a port, sends a given calling user name, and goes into a "while"
     * loop in that:<p>
     *
     * <ol>
     * <li> forward a message from the standard input to the server
     * <li> forward a message from the server to the standard output
     * </ol>
     *
     * @param name   the calling user name
     * @param server a server ip name
     * @param port   a server port
     */
    public ChatClient( String name, String server, int port ) {

       // Create a scoket, register, and listen to the server
       try {
           // Connect to the server
           socket = new Socket( server, port );
           rawIn = socket.getInputStream( );

           // Create an input, an output, and the standard output stream.
           in = new DataInputStream( rawIn );
           out = new DataOutputStream( socket.getOutputStream( ) );
           stdin = new BufferedReader( new InputStreamReader( System.in ) );

           // Send the client name to the server
           out.writeUTF( name );
           while( true ) {
              // If the user types something from the keyboard, read it from
              // the standard input and simply forward it to the srever
              if ( stdin.ready( ) ) {
                  String str = stdin.readLine( );
```

```
                // no more keyboard inputs: the user typed ^d.
                if ( str == null )
                   break;
                out.writeUTF( str );
            }

            // If the server gives me a message, read it from the server
            // and write it down to the standard output.
            if ( rawIn.available( ) > 0 ) {
                String str = in.readUTF( );
                System.out.println( str );
            }
        }
        // Close the connection. That's it.
        socket.close( );
      } catch ( Exception e ) {
        e.printStackTrace( );
      }
    }

    /**
     * Usage: java ChatClient <your_name> <server_ip_name> <port>
     *
     * @param args Receives a client user name, a server ip name, and its port
     *             in args[0], args[1], and args[2] respectively.
     */
    public static void main( String args[] ) {
      // Check # args.
      if ( args.length != 3 ) {
        System.err.println( "Syntax: java ChatClient <your name> " +
                            "<server ip name> <port>" );
        System.exit( 1 );
      }

      // convert args[2] into an integer that will be used as port.
      int port = Integer.parseInt( args[2] );

      // instantiate the main body of ChatClient application.
      new ChatClient( args[0], args[1], port );
    }
}
```

The code is very simple. It checks the number of arguments, and instantiates a ChatClient object that implements the above algorithm. There are some Java specific classes you should be reminded of:

InputSteam	This is a byte-streamed socket input. To read messages from an established socket, you must first obtain this object through getInputStream( ) from the socket. Messages must be bytes.
OutputStream	This is a byte-streamed socket output. To write messages to an established socket, you must first obtain this object through getOutputStream( ) from the socket. Messages must be bytes.
DataInputStream	This filters a given InputStream object, so that you can receive messages in any primitive types other than bytes. The readUTF( ) reads in a string that has been encoded using a modified UTF-8 format.

DataOutputSteam	This filters a given OutputStream object, so that you can send messages in any primitive types other than bytes. The writeUTF( ) writes in a string that has been encoded using a modified UTF-8 format.

B. Chat Server

A chat server program should be invoked with one argument, namely a server port:

[css434@uw1-320-20 hw1]$ java ChatServer 12345

The server program establishes a server socket with a given port and creates a list of client connections that is of course empty at the beginning. Thereafter, the server goes and remains in an infinitve lopp where it repeats the following operations:
(1) Accept a new client socket connection if there is any,
(2) Read the client name with readUTF( ),
(3) Add this connection into the list of existing client connections.
(4) For each connection of the list,
    (a) Receive a new message with readUTF( ) if there is any,
    (b) Add the client name in front of the message received.
    (c) Write this message to all clients through all connections of the list. Use writeUTF( ).
    (d) Check if any errors have occurred when reading and writing a message. If so, remove this connection from the list.

The server program is not shown, because it is your assignment. ☺ The following gives some programming hints:

(1) Java ServerSocket Class
    Use the ServerSocket(int port) constructor when instantiating a server socket. The accept( ) is a blocking call as default. However, we do not want to be blocked upon accept( ). Use setSoTimeout(500) so that the accept( ) can return in 500msec. When calling accept( ), you have to use try{ } catch(SocketTimeoutExeception e) { }. Whenever returning from accept( ), check if the return value has a reference to a new socket or null. If it has a reference, you got a new connection and thus can add it to a list of connections.

(2) Connection Class
    I recommend you should design your Connection class whose instance can maintain a client socket connection and provide readMessage( ) and writeMessage( ) functions. They are used to read a new message with readUTF( ) and write a message with writeUTF( ). When an error has occurred in a message read and/or a write, record this error, so that you can delete this connection later.

    Again, readUTF( ) is a blocking call as default. Make sure that the socket has some data to read through readUTF( ) before actually calling readUTF( ). For this

purpose, Use the available( ) of the InputStream( ) class that you should obtain from the socket.

C. Statement of Work

Design and code ChatServer.java according to the above server specification. Compile your ChatServer.java and the professor's ChatClient.java (located in the uw1-320: ~css434/hw1/ directory). Run them as follows:

(1). Choose four different machines in uw1-320.
(2). Start ChatServer first on one of these machines.
(3). Run ChatClient on each of the other three machines with a different client name such as mickey, mini, and goofy or whatever you like.
(4). Test if a client message is broadcast through the server to all the clients.
(5). Take an execution snapshot. (Type "import –window root X.jpeg" at an xterm, show X.jpg with firefox, and print it out to uw1-320-p1.)

Part 2. Chats in Causal Order

A. Serverless Chat Client

This program named Chat.java does not use a central server. All computing nodes have to do is just run this chat client with the same arguments:

[css434@uw1-320-20 hw1]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22

The first argument is the IP port, (i.e. the last five digits of your student ID) used to establish a TCP connection to each remote computing node. The following arguments are a list of computing nodes that participate in the same chatting session. The order of computing nodes must be the same at each node.

The main function verifies the arguments and instantiates a Chat class. The constructor first creates a complete TCP network among all computing nodes. To broadcast a local message to all the other chat members, we repeat sending the message to each of ObjectOutputStreams, outputs[*i*] where 0 <= *i* < #hosts, (i.e., hosts.length). To receive a message from a remote chat member, we read a message from an ObjectInputStream, inputs[*i*]. The following shows the message broadcasting and receiving portion of this serverless client program, (Chat.java):

```
// now goes into a chat
while ( true ) {
    // read a message from keyboard and broadcast it to all the others.
    if ( keyboard.ready( ) ) {
        // since keyboard is ready, read one line.
        String message = keyboard.readLine( );
        if ( message == null ) {
            // keyboard was closed by "^d"
            break; // terminate the program
        }
        // broadcast a message to each of the chat members.
        for ( int i = 0; i < hosts.length; i++ )
            if ( i != rank ) {
                // of course I should not send a message to myself
```

```
                outputs[i].writeObject( message );
                outputs[i].flush( ); // make sure the message was sent
            }
        }

        // read a message from each of the chat members
        for ( int i = 0; i < hosts.length; i++ ) {
            // to intentionally create a misordered message deliveray,
            // let's slow down the chat member #2.
            try {
                if ( rank == 2 )
                    Thread.currentThread( ).sleep( 5000 ); // sleep 5 sec.
            } catch ( InterruptedException e ) {}

            // check if chat member #i has something
            if ( i != rank && indata[i].available( ) > 0 ) {
                // read a message from chat member #i and print it out
                // to the monitor
                try {
                    String message = ( String )inputs[i].readObject( );
                    System.out.println( hosts[i] + ": " + message );
                } catch ( ClassNotFoundException e ) {}
            }
        }
    }
```

B. Creating a miss-ordered delivery

Look at the code snippet in blue above. It delays the third chat member, (i.e., rank 2) for 5 seconds. This prevents the third chat member from receiving messages from the others in the causal order. For instance, let's assume that the first chat member writes "Go skiing?" and thereafter the second chat member responds "Yes".

[css434@uw1-320-20 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 0, localhost = uw1-320-20
accepted from uw1-320-21.uwb.edu
accepted from uw1-320-22.uwb.edu
Go skiing?
Uw1-320-21: Yes

css434@uw1-320-21 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 1, localhost = uw1-320-21
accepted from uw1-320-22.uwb.edu
connected to uw1-320-20
uw1-320-21: Go skiing?
Yes

css434@uw1-320-22 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 1, localhost = uw1-320-21
connected to uw1-320-22
connected to uw1-320-21
uw1-320-21:Yes
uw1-320-20: Go skiing?

As shown above, the third chat member (at uw1-320-22) mistakenly receives “Yes” first and “Go skiing?” which is not the causal order.

C. Statement of Work

Insert additional code into Chat.java to implement the causal ordering so that all chat members can receive a message before its subsequent messages generated by the third party. For instance, “Go skiing?” and “Yes” messages should be printed out in this order at all computing nodes like:

```
[css434@uw1-320-20 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 0, localhost = uw1-320-20
accepted from uw1-320-21.uwb.edu
accepted from uw1-320-22.uwb.edu
Go skiing?
Uw1-320-21: Yes

css434@uw1-320-21 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 1, localhost = uw1-320-21
accepted from uw1-320-22.uwb.edu
connected to uw1-320-20
uw1-320-21: Go skiing?
Yes

css434@uw1-320-22 hw1.new]$ java Chat 12345 uw1-320-20 uw1-320-21 uw1-320-22
port = 12345, rank = 1, localhost = uw1-320-21
connected to uw1-320-22
connected to uw1-320-21
uw1-320-20: Go skiing?
uw1-320-21:Yes
```

In your implementation, each computing node must maintain a vector of stamps, where each element *i* counting send/receive events at node ***i***. Before broadcasting a new message, the local host should increment its corresponding element of the vector. It should send the message as well as the vector. Whenever receiving a new message from a sender *j*, your have to also receive this sender’s vector and compare its contents with the local vector contents. The comparison must be achieved as follows:

For each element *i*, if *i* == *j*, the sender’s vector[*i*] must be one larger than the local host’s vector[*i*]. Otherwise the sender’s vector[*i*] must be smaller than or equal to the local host’s vector[*i*].

If these conditions are satisfied, you can accept and print out the message as incrementing the local host’s vector[*j*]. Otherwise, keep this message and vector in a waiting list, and check it later when you receive another message.

Note that you must not modify any original code statements. All you can do is add your new code into Chat.java.

After implementing your causal-ordered chat client, run it as follows:

(1). Choose three different machines in uw1-320.
(2). Run Chat on each of the other three machines
(3). Test if they exchange messages in the causal order with the above scenario: "Go skiing?" and "Yes".
(4). Take an execution snapshot. (Type "import –window root X.jpeg" at an xterm, show X.jpg with firefox, and print it out to uw1-320-p1.)

3. What to Turn in

The homework is due at the beginning of class on the due date. You have to turn in the following materials in a hard copy. No email submission is accepted.

Criteria	**Percentage**
Documentation of your algorithm including explanations and illustrations in one or two pages. (1) Part 1 Algorithm (consistent ordering): 2.5pts (2) Part 2 Algorithm (causal ordering): 2.5pts	5pts (25%)
Source code that adheres good modularization, coding style, and an appropriate amount of comments. For each of part 1 and part 2: well-organized and correct code receives 2.5pts, messy yet working code receives 2pts, code with bugs receives 1.5pts, and incomplete code receives 1pt.	5pts (25%)
Execution output such as a snapshot of your display/windows or contents of standard output redirected to a file. For each of part 1 and part 2: a correct output receives 2.5pts, the one with minor errors receives 2pts, and an incomplete output receives 1.5pt.	5pts (25%)
Discussions about the efficiency of your algorithm and possible performance/functional improvement in one page. (1) Part 1 Algorithm (consistent ordering): 2.5pts (2) Part 2 Algorithm (causal ordering): 2.5pts	5pts (25%)
Total	20pts (100%)