Kai Zhang

CSS 434

Professor Fukuda

1. Documentation including explanations and illustrations

Part 1: Consistent ordering:

A diagram of a process

Description automatically generated

In consistent ordering, the Server listens on a given port, and listens for incoming connections from clients, then add the connections to a list. Where the server would use a for each loop to listen for any messages sent from the clients, and purges any dead clients. If a client writes a message, the server would use a for each loop to write back the messages. This algorithm is very simple and does not require ordering algorithms. However, it does not scale well due to the nested for loops, and will run at O(N^2) time.

Part 2. Casual ordering:

A diagram of a flowchart

Description automatically generated

In casual ordering, there isn’t a central server, instead order is kept through a vector that is passed around the clients. The vector tracks the recency of the messages to keep order. If the recency is wrong, then it means there is another message that needs to be received, and the program will check back later and put the message in a queue.

1. Discussions about the efficiency of algorithm, possible improvements:
   1. Consistent ordering:
      1. Efficiency: It does not scale well due to the nested for loops, and will run at O(N^2) time. And will put a lot of strain on the server if multiple users enter. And latency will develop and impeding the timely delivery of messages
      2. Improvements: Should add features such as time stamps and Global Sequencer on servers to ensure correct delivery of messages.
   2. Casual ordering:
      1. Efficiency: Casual ordering can balance load and scale better due to its P2P nature, but due to the added Metadata, the packets becomes larger. Casual ordering will buildup queues in networks with more congestions.
      2. As networks become more congested, more messages will be delivered incorrectly, therefore a total order may be needed to be created in order to make sure all peers have the same version of message history.

Execution snapshot:A black screen with white text

Description automatically generated

A black and white image of a black screen

Description automatically generated

Source Code:

ChatServer.java:

// Make sure to change the file name into ChatServer.java

/\*\*

 \* ChatServer.java:<p>

 \*

 \*

 \* @author  Munehiro Fukuda (CSS, University of Washington, Botheel)

 \* @since   1/23/05

 \* @version 2/5/05

 \*/

import java.net.\*;          // for Socket and ServerSocket

import java.io.\*;           // for IOException

import java.util.\*;         // for Vector

public class ChatServer {

    // a list of existing client connections to be declared here.

    ArrayList<Connection> connList = new ArrayList<Connection>();

    /\*\*

     \* Creates a server socket with a given port, and thereafter goes into

     \* an infinitive loop where:<p>

     \* <ol>

     \* <li> accept a new connection if there is one.

     \* <li> add this connection into a list of existing connections

     \* <li> for each connection, read a new message and write it to all

     \*      existing connections.

     \* <li> delete the connection if it is already disconnected.

     \* </ol>

     \*

     \* @param port an IP port

     \*/

    public ChatServer( int port ) {

        try{

            ServerSocket server = new ServerSocket(port);

            while( true ) {

                try {

                    // Create a sersver socket

                    // ServerSocket server = ....

                    server.setSoTimeout( 500 ); //will be blocked for 500ms upon accept

                    Socket client = server.accept();// accept a new connection

                    if(client == null){   // if this connection is not null

                        continue;

                        //add the new connection into a list of existing connections

                    }else{

                        Connection conn = new Connection(client);

                        connList.add(conn);

                        System.out.println("Added connection: " + conn.name);

                    }

                } catch ( SocketTimeoutException e) {

                    //System.out.println("Socket exception");

                }

                // for each connection, read a new message and write it to all

                // existing connections

                for (Connection connection:connList) {

                    String message = connection.readMessage();

                    //System.out.println("Broadcast: " + message);

                    if(message == null){// read a new message if exist.

                        continue;// make sure that this read won't be blocked.

                    }else{

                        for(Connection allconns:connList){

                            // if you got a message, write it to all connections.

                            if(allconns.isAlive() == true){

                                allconns.writeMessage(message);

                            }else{

                                connList.remove(allconns);

                                // delete this connection if the client disconnected it

                            }

                        }

                    }

                }

            }

                //server.close();

        }catch (IOException e){

            e.printStackTrace();

        }

    }

    /\*\*

     \* Usage: java ChatServer <port>

     \*

     \* @param args a String array where args[0] includes port.

     \*/

    public static void main( String args[] ) {

        // check if args[0] has port

        if ( args.length != 1 ) {

            System.err.println( "Syntax: java Chatserver <port>" );

            System.exit( 1 );

        }

        // start a chat server

        new ChatServer( Integer.parseInt( args[0] ) );

    }

    /\*\*

     \* Represents a connection from a different chat client.

     \*/

    private class Connection {

        private Socket socket;        // a socket of this connection

        private InputStream rawIn;    // a byte-stream input from client

        private OutputStream rawOut;  // a byte-stream output to client

        private DataInputStream in;   // a filtered input from client

        private DataOutputStream out; // a filtered output to client

        private String name;          // a client name

        private boolean alive;        // indicate if the connection is alive

        /\*\*

         \* Creates a new connection with a given socket

         \*

         \* @param client a socket representing a new chat client

         \*/

        public Connection( Socket client ) {

            socket = client;

            try {

                rawIn = socket.getInputStream();

                rawOut = socket.getOutputStream();

                in = new DataInputStream(rawIn);

                out = new DataOutputStream(rawOut);

                name = in.readUTF();

                alive = true;

                System.out.println(name);

            } catch (IOException e) {

                // TODO: handle exception

                e.printStackTrace();

            }

            // from socket, initialize rawIn, rawOut, in, and out.

            // the first message is a client name in unicode format

            // upon a successful initialization, alive should be true.

        }

        /\*\*

         \* Reads a new message in unicode format and returns it with this

         \* client's name.

         \*

         \* @return a unicode message with the client's name

         \*/

        public String readMessage( ) {

            try {

                if(rawIn.available() > 0){

                    String toReturn;

                    toReturn = in.readUTF();

                    System.out.println("Readmsg: " + toReturn);

                    return name + ": " + toReturn;

                }else{

                    //System.out.println("Unavailible readmsg");

                    return null;

                }

            } catch (IOException e) {

                e.printStackTrace();

                return null;

            }

            // read a message if it's available.

            // don't block. use available( ).

            // if it returns a positive number, you can read it.

            // otherwise, skip reading.

        }

        /\*\*

         \* Writes a given message through this client's socket.

         \*

         \* @param message a String to write to the client

         \*/

        public void writeMessage( String message ) {

            try {

                // write a message

                System.out.println("Output: " + message);

                out.writeUTF(message);

                out.flush();

                // use flush( ) to send it immediately.

            } catch (IOException e) {

                alive = false;

                e.printStackTrace();

            }

            // if an exception occurs, you can identify that this connection

            // was gone.

        }

        /\*\*

         \* Checks if this connection is still live.

         \*/

        public boolean isAlive( ) {

            // if the connection was broken, return false.

            return alive;

        }

    }

}

Chat.java:

import java.net.\*;  // ServerSocket, Socket

import java.util.ArrayList;

import java.util.Vector;

import java.io.\*;   // InputStream, ObjectInputStream, ObjectOutputStream

public class Chat {

    // Each element i of the follwoing arrays represent a chat member[i]

    private Socket[] sockets = null;             // connection to i

    private InputStream[] indata = null;         // used to check data from i

    private ObjectInputStream[] inputs = null;   // a message from i

    private ObjectOutputStream[] outputs = null; // a message to i

    private int[] vector = null;

    private Vector<int[]> vec\_queue = new Vector<int[]>();              // maintains vector stamps from the others

    private Vector<String>  msg\_queue = new Vector<String>( );      //

    private Vector<Integer> src\_queue = new Vector<Integer>( );

    /\*\*

     \* Is the main body of the Chat application. This constructor establishes

     \* a socket to each remote chat member, broadcasts a local user's message

     \* to all the remote chat members, and receive a message from each of them.

     \*

     \* @param port  IP port used to connect to a remote node as well as to

     \*              accept a connection from a remote node.

     \* @param rank  this local node's rank (one of 0 through to #members - 1)

     \* @param hosts a list of all computing nodes that participate in chatting

     \*/

    public Chat( int port, int rank, String[] hosts ) throws IOException {

        // print out my port, rank and local hostname

        System.out.println( "port = " + port + ", rank = " + rank +

                    ", localhost = " + hosts[rank] );

        // create sockets, inputs, outputs, and vector arrays

        sockets = new Socket[hosts.length];

        indata = new InputStream[hosts.length];

        inputs = new ObjectInputStream[hosts.length];

        outputs = new ObjectOutputStream[hosts.length];

        int[] vector = new int[hosts.length]; //this is the timestamp

        // establish a complete network

        ServerSocket server = new ServerSocket( port );

        for ( int i = hosts.length - 1; i >= 0; i-- ) {

            if ( i > rank ) {

            // accept a connection from others with a higher rank

            Socket socket = server.accept( );

            String src\_host = socket.getInetAddress( ).getHostName( );

            // find this source host's rank

            for ( int j = 0; j < hosts.length; j++ )

                if ( src\_host.startsWith( hosts[j] ) ) {

                    // j is this source host's rank

                    System.out.println( "accepted from " + src\_host );

                    // store this source host j's connection, input stream

                    // and object intput/output streams.

                    sockets[j] = socket;

                    indata[j]= socket.getInputStream( );

                    inputs[j] =

                        new ObjectInputStream( indata[j] );

                    outputs[j] =

                        new ObjectOutputStream( socket.getOutputStream( ));

                }

            }

            if ( i < rank ) {

                // establish a connection to others with a lower rank

                sockets[i] = new Socket( hosts[i], port );

                System.out.println( "connected to " + hosts[i] );

                // store this destination host j's connection, input stream

                // and object intput/output streams.

                outputs[i]

                    = new ObjectOutputStream( sockets[i].getOutputStream( ) );

                indata[i] = sockets[i].getInputStream( );

                inputs[i]

                    = new ObjectInputStream( indata[i] );

            }

        }

        // create a keyboard stream

        BufferedReader keyboard

            = new BufferedReader( new InputStreamReader( System.in ) );

        // 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.

            vector[rank] += 1; // v(gi)[i] = v(gi) + 1

            for ( int i = 0; i < hosts.length; i++ )

                if ( i != rank ) {

                    // of course I should not send a message to myself

                    outputs[i].writeObject( vector ); //sends vector

                    outputs[i].writeObject( message ); //sends 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 ) {}

                checkqueue(i, hosts, vector);

                // 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 {// insert filter here

                        //Read inputs[i].readObject( ). This will be a vector.

                        int[] sendVector = (int[])inputs[i].readObject( );

                        //read inputs[i].readObject( ) one more time. This will be an actual message.

                        String message = (String)inputs[i].readObject( );

                        //check if this received vector and my vector. If this received vector is ready to accept

                        boolean print = false;

                        for(int j = 0; j < vector.length; j++){ //For each element

                            if(i == j){

                                if(sendVector[i]+1 == vector[i]){

                                    print = true;

                                    // sender’s vector[i] must be one larger than the local host’s vector[i].

                                }

                            }else{ //i!== j

                                if(sendVector[j] <= vector[j]){

                                    print = true;

                                    // sender’s vector[i] must be smaller than or equal to the local host’s vector[i].

                                }

                            }

                        }

                        if(print){

                            System.out.println( hosts[i] + ": " + message );

                            vector[i] += 1;

                        }else{

                        //Otherwise this received vector and message are not ready to print out. Therefor,

                        //Enqueue this received vector into vec\_queue.

                            vec\_queue.addElement(sendVector);

                        //Enqueue the receive message into msg\_queue.

                            msg\_queue.add(message);

                        //Enguque Interger( i ), (i.e., this source process ID) into src\_queue.

                            src\_queue.add(i);

                        }

                    } catch ( ClassNotFoundException e ) {}

                }

            }

        }

    }

    /\*\*

     \* Checkqueue

     \* -----------------------------

     \* this function would take in the host's ID, the list of hostnames

     \* as well as the vector to update it

     \* @param host

     \* @param hostnames

     \* @param vector

     \*/

    private void checkqueue(int host, String[] hostnames, int[] vector){

        if(src\_queue.isEmpty() == false){

            for(int i = 0; i < vec\_queue.size(); i++){ //checking the waitlist to see if any of them are okay to send

                for(int j = 0; j < vector.length; j++){

                    if(host == j){

                        if(vec\_queue.get(i)[host]+1 == vector[host]){

                            System.out.println(hostnames[src\_queue.get(i)] + ":" + msg\_queue.get(i));

                            vector[host] += 1;

                            msg\_queue.remove(i);

                            vec\_queue.remove(i);

                            src\_queue.remove(i);

                            i--;

                        }

                    }else{

                        if(vec\_queue.get(i)[j] <= vector[j]){

                            System.out.println(hostnames[src\_queue.get(i)] + ":" + msg\_queue.get(i));

                            vector[host] += 1;

                            msg\_queue.remove(i);

                            vec\_queue.remove(i);

                            src\_queue.remove(i);

                            i--;

                        }

                    }

                    if(src\_queue.isEmpty()){

                        break;

                    }

                }

            }

        }

    }

    /\*\*

     \* Is the main function that verifies the correctness of its arguments and

     \* starts the application.

     \*

     \* @param args receives <port> <ip1> <ip2> ... where port is an IP port

     \*             to establish a TCP connection and ip1, ip2, .... are a

     \*             list of all computing nodes that participate in a chat.

     \*/

    public static void main( String[] args ) {

        // verify #args.

        if ( args.length < 2 ) {

            System.err.println( "Syntax: java Chat <port> <ip1> <ip2> ..." );

            System.exit( -1 );

        }

        // retrieve the port

        int port = 0;

        try {

            port = Integer.parseInt( args[0] );

        } catch ( NumberFormatException e ) {

            e.printStackTrace( );

            System.exit( -1 );

        }

        if ( port <= 5000 ) {

            System.err.println( "port should be 5001 or larger" );

            System.exit( -1 );

        }

        // retireve my local hostname

        String localhost = null;

        try {

            localhost = InetAddress.getLocalHost( ).getHostName( );

        } catch ( UnknownHostException e ) {

            e.printStackTrace( );

            System.exit( -1 );

        }

        // store a list of computing nodes in hosts[] and check my rank

        int rank = -1;

        String[] hosts = new String[args.length - 1];

        for ( int i = 0; i < args.length - 1; i++ ) {

            hosts[i] = args[i + 1];

            if ( localhost.startsWith( hosts[i] ) )

            // found myself in the i-th member of hosts

            rank = i;

        }

        // now start the Chat application

        try {

            new Chat( port, rank, hosts );

        } catch ( IOException e ) {

            e.printStackTrace( );

            System.exit( -1 );

        }

    }

}