Network Computing Lecture 6

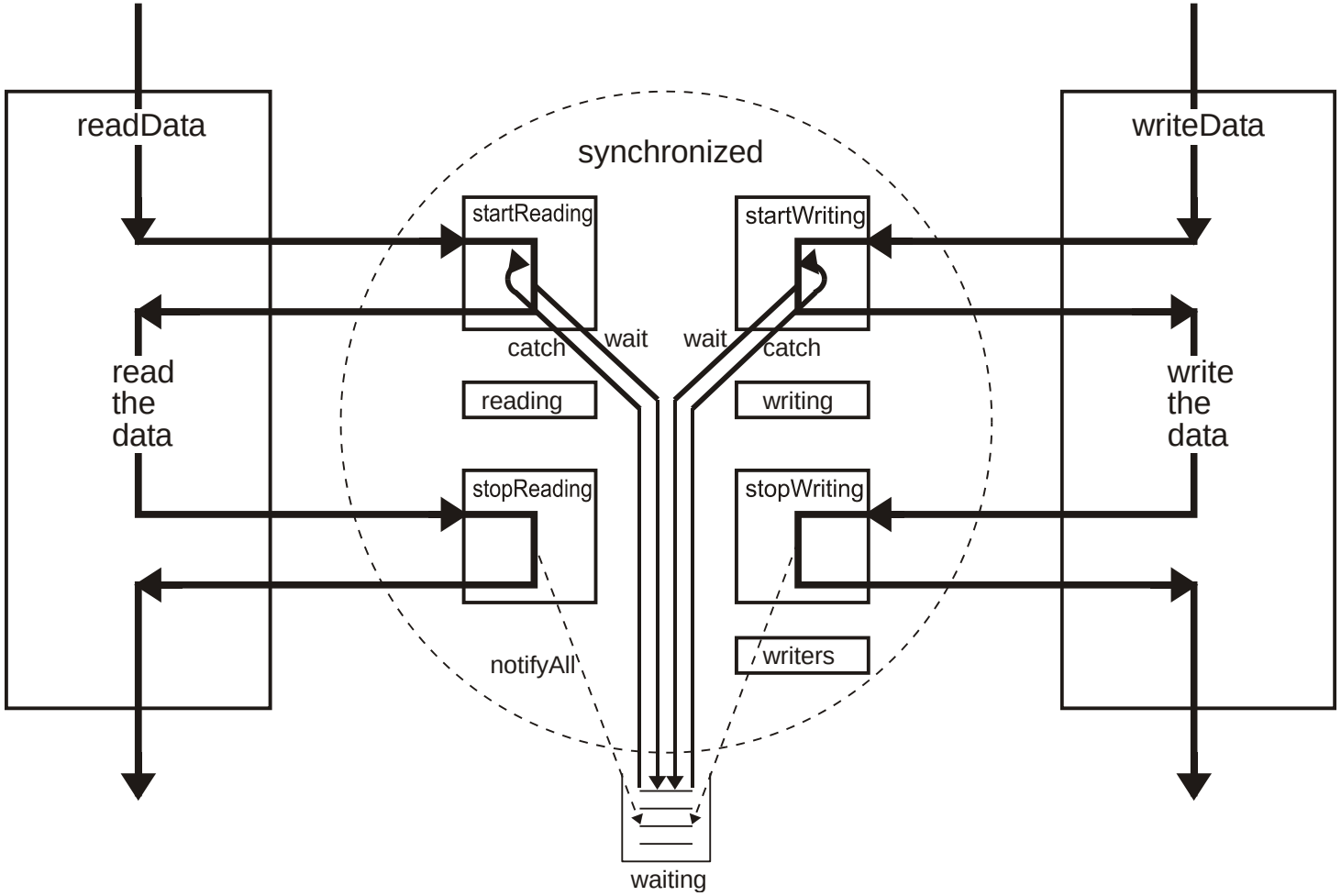
Professor Louise E. Moser

Spring 2012

More on Sockets and Threads

- As we have seen in Lecture 5, Java's **synchronized** methods permit **only a single thread** in the **monitor** (**critical section**) at a time
- However, often you need a monitor such that:
  - Multiple reader threads can read concurrently
  - Only one writer thread can write at a time
  - No reader thread can read while a writer thread is writing
- For example, if you have a calendar object, multiple clients should be able to read the calendar concurrently but not while the server that maintains the calendar is updating it
- Java's synchronized methods provide a monitor for only a single reader or writer. However, they can be used to create a more general monitor that accepts multiple readers

Multiple Readers/Single Writer Monitor



Multiple Readers/Single Writer Monitor

```
int reading = 0;        // Number of readers reading (0 or more)
int writing = 0;        // Number of writers writing (0 or 1)
int writers = 0;        // Number of writers writing or waiting to write
                        // A writer (writing or waiting) prevents new readers from reading

public void readData(){
 startReading();        // Ask for permission to read
 ...                    // Read the data
 stopReading();         // Signal that you are done
}

public void writeData(){
 startWriting();        // Ask for permission to write
 ...                    // Write the data
 stopWriting();         // Signal that you are done
}
```

Multiple Readers/Single Writer Monitor

```
public synchronized void startReading(){  // Enter synchronized method  before reading
 while (writers > 0) {                    // Check if a writer is writing or waiting to write
   try {
     wait();                              // If so, wait
   }
   catch (InterruptedException e) {}      // Wakeup
 }                                        // Go back and try again
 reading = reading + 1;                   // I'm now reading
}

public synchronized void stopReading(){  // Enter synchronized method after reading
  reading = reading –  1;                  // I'm no longer reading
  if (reading == 0) then                  // Notify waiting writer if no readers are reading
    notifyAll();
}
```

Multiple Readers/Single Writer Monitor

```
public synchronized void startWriting(){  // Enter synchronized method before writing
  writers = writers + 1;                  // I want to write; excludes future readers
  while ((writing > 0) or                 // Check if writer is writing or
         (reading > 0)) {                 //  readers are reading
   try {
     wait();                              // If so, wait
   }
   catch (InterruptedException e) {}      // Wakeup
 }                                        // Go back and try again
 writing = writing + 1;                   // I'm now writing; excludes other writers
}

public synchronized void stopWriting(){  // Enter synchronized method after writing
  writers = writers – 1;                  // I'm no longer writing or waiting to write
  writing = writing – 1;                  // I'm no longer writing; no writers are writing
  notifyAll();                            // Notify other readers and writers
}
```

Use of Multiple Readers/Single Writer Monitor

- ***MultiJabberServer2*** is an echo server that can serve several clients **simultaneously**
- It consists of two parts:
  - *ServeOneJabber2* is a thread that serves a single client
    - It reads a message and echoes the content back to the client
    - It also reads the *visitorCount* and sends that back to the client
  - *MultiJabberServer2* waits on the public server port for new clients to connect to it. For each such client,
    - It obtains a new socket for the connection to the client and spawns a new *ServeOneJabber2* thread
    - It also increments the *visitorCount*, which keeps track of the number of clients that have connected

The main() method of MultiJabberServer2

- The *main()* method opens a *ServerSocket s* and then enters an infinite loop waiting for clients to connect

```
while(true) {
 Socket socket = s.accept();
 incrementVisitorCount();
 try {
    new ServeOneJabber2(socket);
 }
 catch(IOException e) {}
}
```

- When a client opens a connection, *s.accept()* returns the socket allocated to that client
- Next, the *visitorCount* is incremented
- Then, a new *ServeOneJabber2* thread is created and initialized with the new socket

The incrementVisitorCount() Method

- This method must have **exclusive access** to *visitorCount*, while it is incrementing it, to avoid erroneous results (actually, this example is so simple that an error will not occur)
- *visitorCount* is declared and initialized to zero by *MultiJabberServer2*

```
public int incrementVisitorCount() {
  private int i;
  startWriting();          // Asks for permission to write the data
                           // It now has exclusive access to do so
  i = visitorCount + 1;
  visitorCount = i;        // Writes the data
  stopWriting();           // Signals that it is done writing
                           // so that others can access the data
  return i;
}
```

Declaring the ServeOneJabber2 Thread

- This thread extends the *Thread* class and overwrites its *run()* method

```
class ServeOneJabber2 extends Thread {
    private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
```

Initializing the ServeOneJabber2 Thread

```
public ServeOneJabber2(Socket s) // Constructor
  throws IOException {
 socket = s;
  in = new BufferedReader(
               new InputStreamReader(socket.getInputStream()));
  out = new PrintWriter(
                new BufferedWriter(
                      new OutputStreamWriter(socket.getOutputStream()), true);
start();       // Calls the run() method
}
```

- The parameter *s* is used to initialize *socket* that connects to the client and to open input and output streams to the client
- *start()* launches the *run()* method

ServeOneJabber2's run() Method

```
public void run() {
  private int i;
  try {
    while (true) {
      String str = in.readLine();
      if (str.equals("END")) break;
         ....
      out.println(str);
      i = getVisitorCount();
      out.println(i + " visitors so far");
    }
  }
  catch ...
```

- Within an infinite loop, the thread
  - Reads a string from the input stream
  - Sends it to the output stream, along with the number of visitors that have connected to the server so far
- Until it reads an *"END"* string

The getVisitorCount() Method

- Access to the *visitorCount* data must be protected against concurrent reading and writing to avoid erroneous results (again, this example is so simple that an error will not occur)

```
public int getVisitorCount() {
  private int i;
  startReading();          // Ask for permission to read the data
                           // The data is now protected against writers
  i = visitorCount;        // Read the data
  stopReading();           // Signal that it is done reading
                           // so that others can access the data
  return i;
}
```

Datagram Sockets and Packets

- **Datagram Packet -** Packet of data sent by the User Datagram Protocol (UDP)
- **Datagram Socket** – Socket over which a datagram packet is sent by UDP
- UDP is **connectionless**, i.e., it does not require you to establish a connection first before sending a datagram
- UDP is **unreliable**, i.e., it does not guarantee the arrival, arrival time, or content of the datagrams
- The *DatagramSocket* and *DatagramPacket* classes in *java.net* allow you to send and receive datagram packets through a datagram socket

Ex: The Quote Server and Client

- Client sends a datagram packet to the server, requesting a quote from the server, and waits for the server to send a datagram packet in response
- Server gets a quote from its file of quotations, creates a datagram packet containing the quote, and sends the datagram packet to the client
- Client extracts the quote from the datagram packet it received from the server, and displays the quote

The QuoteServer Class

- The *main()* method of *QuoteServer* creates a new *QuoteServerThread* and starts it running

```
public class QuoteServer {
    public static void main(String[] args) {
      new QuoteServerThread().start();
    }
}
```

The QuoteServerThread

- The *QuoteServerThread* creates a *DatagramSocket* and a *BufferedReader*, and waits for a request from a client, to arrive over the datagram socket
- In response to a client's request, the Q*uoteServerThread*
  - Gets a quote from the file
  - Puts it in a datagram packet
  - Sends it on the datagram socket to the client

Creating a Datagram Socket and a Buffered Reader

```
socket = new DatagramSocket(4445);
try {
  in = new BufferedReader(new FileReader("one-liners.txt"));
}
catch (FileNotFoundException e) {
  System.err.println("Couldn't open quote file. " + "Serving time instead.");
}
```

- Creates a new *DatagramSocket* on port 4445, on which *QuoteServerThread* listens for, and responds to, client requests
- Creates a *BufferedReader* for a **file** named *one-liners.txt*, which contains a list of quotes, each on a separate line

Receiving a Request from the Client

```
byte[] buf = new byte[256];
DatagramPacket packet =
      new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

- The array *buf* is used by the server to create a *DatagramPacket*
- The *packet* is used by the server to receive a datagram packet from a client on the datagram socket
- The *receive()* method waits until the server receives a packet from a client on the datagram socket
  - If it does not receive a packet, the server does nothing but wait

Constructing a Response to the Client

- When the server receives a request from a client for a quote, it constructs a response

```
String dString = null;
if (in == null)
   dString = new Date().toString();
else
   dString = getNextQuote();
buf = dString.getBytes();
```

- If the quote file is empty, the server serves up the time of day
- Otherwise, it gets the next quote from the opened file and converts the string to an array of bytes

Sending the Response to the Client

InetAddress ***address*** = ***packet.getAddress();***
int ***port*** = ***packet.getPort();***
responsePacket = new DatagramPacket(buf, buf.length, ***address****,* ***port****);*
socket.send(responsePacket);

- The server gets the Internet address and port number from the packet that it received from the client
- It creates a new *DatagramPacket* for sending the packet, containing its response, over the datagram socket to the client
  - The packet contains a byte array that holds
    1. Server's message in the array *buf*
    2. Length of the array *buf*
    3. Client's Internet address
    4. Client's port number
- It sends the datagram packet containing its response to the client

The QuoteClient Class

- The Quote Client sends a request to the Quote Server, waits for the response, and displays the response to the standard output
- The *main()* of Quote Client uses a command-line argument: the name of the server computer on which Quote Server is running

```
if (args.length != 1) {
  System.out.println("Usage: java QuoteClient <hostname>");
  return;
}
```

- This code assumes that the user knows the host name of the server and knows to type:
  java QuoteClient hostname
- If the user screws up, the program supplies the prompt
  Usage: java QuoteClient <hostname>

Creating a DatagramSocket

DatagramSocket socket = new DatagramSocket();

- Creates a *DatagramSocket* and binds it to any available local port at the client

Sending the Request to the Server

```
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(args[0]);
DatagramPacket packet = new DatagramPacket(buf, buf.length,
                                           address, 4445);
socket.send(packet);
```

- Gets the Internet address for the server computer, named on the command line, from DNS and puts it in the address field of the packet header, along with the port number 4445 of the Quote Server
- The *DatagramPacket* has an empty byte array *buf* for the payload, because it's a simple request to the server for information
- The Internet address and port number of the client, that the server needs to reply to the client, are automatically part of the packet header
- The client then sends the datagram packet, containing its request, on the datagram socket to the server

Receiving and Displaying the Response from the Server

```
DatagramPacket responsePacket = new DatagramPacket(buf, buf.length);
socket.receive(responsePacket);
String received = new String(responsePacket.getData());
System.out.println("Quote of the Moment: " + received);
```

- The client creates a *DatagramPacket* which it uses to receive the response from the server
- The *receive()* method waits for a datagram packet to arrive on the datagram socket from the server
- When the client receives the response packet from the server,
  - It uses *getData()* to retrieve the data from the response packet and then converts the data to a string
  - Then it displays the data

Avoiding Waiting Forever

- Because UDP does not provide any delivery guarantees, a client will wait forever if the client's request or the server's response is lost
- To avoid this problem, a client typically sets a **timer** so that it doesn't wait forever
- If it doesn't receive a response before the timer goes off, the client retransmits its request
- To specify a **timeout**, you can use

  setSoTimeout(int interval);

  where the timeout interval is in milliseconds
- For example,

  DatagramSocket socket = new DatagramSocket();

  socket.setSoTimeout(1000);

Summary: What You Need to Do

- Write the *QuoteServer* and *QuoteClient* programs
- Create the file *one-liners.txt* containing the list of quotes on the server computer A
- Compile the *QuoteServer* and *QuoteClient* programs, using the Java compiler
- Start the server program running first on computer A, using the Java interpreter and specifying the *QuoteServer* class name
- After you have started the server on computer A, run the client program on computer B, using the Java interpreter, specifying the *QuoteClient* class name and giving the name of computer A as the command-line argument
- After the client sends a request and receives a response, you should see output similar to this:

Quote of the Moment: Good Java programming is 99% sweat and 1% coffee.