

G52CON: Concepts of Concurrency

Lecture 15 Distributed Processing in Java I

Brian Logan
School of Computer Science & IT
bsl@cs.nott.ac.uk

Outline of this lecture

- message passing and `java.net`
- ports & sockets
- examples:
 - `KnockKnockServer`
 - `KKMultiServer`
 - `ChatServer`

Message passing

Processes communicate by sending and receiving messages using special message passing primitives which include synchronisation:

- **send** *destination message*: sends *message* to another process *destination*
- **receive** *source message*: indicates that a process is ready to receive a message *message* from another process *source*

Synchronising communication

If a process tries to **receive** a message before one has been sent, it will block until there is a message for it to read.

The differences are mainly in the behaviour of the **sending** process:

- asynchronous communication: the sending process continues without waiting for the message to be received, e.g., Unix sockets, `java.net`
- synchronous communication: the sending process is delayed until the corresponding receive is executed, e.g., CSP, occam
- remote invocation: the sending process is delayed until a reply is received, e.g., RPC (`java.rmi`), Extended Rendezvous

java.net

The package `java.net` supports *asynchronous* message passing:

- naming is indirect and (a)symmetrical
- communication is one way—a process can only send or receive on a given channel in a single operation
- receiving is synchronous
- sending is asynchronous

Ports

Application processes are identified by their transport protocol (TCP or UDP) and *port number(s)* (16 bit values):

- e.g., all systems that offer Telnet do so on port 23
- the source port number identifies the process that sent the data and destination port number identifies the process that will receive the data
- dynamically allocated ports are assigned when needed, e.g., for Telnet connections the source port is assigned a dynamically allocated port number and port 23 is used for the destination port
- it is the *pair* of port numbers that uniquely identifies each network connection

Sockets

The combination of an IP address and a port number is called a *socket*:

- a socket is one end of a two-way communication link between two or more processes
- each pair of sockets provides 2 unidirectional channels (streams)
- communication is one-way—a process can only send or receive on a given channel (stream) in a single operation
- *cf* remote invocation, where a process both sends and receives on the same channel in a single method invocation

Transport layer protocols

`java.net` provides support for two transport layer protocols:

- TCP provides reliable, ordered, buffered (one-way) communication between two processes, e.g., HTTP, FTP, Telnet etc. (`Socket`, `ServerSocket`)
- UDP sends independent packets of data (datagrams) from one process to another with no guarantees about reliability or ordering, e.g., ping, time services etc. (`DatagramSocket`, `MulticastSocket`)

java.net TCP

- the `Socket` and `ServerSocket` classes provide reliable, ordered, buffered communication between two processes;
- communication is point-to-point—each `Socket` and `ServerSocket` provide two unidirectional byte streams, an `InputStream` and an `OutputStream`;
- naming is indirect and (a)symmetrical—clients name the host and port to which they wish to connect, servers accept connections from any client on that port and then create a new socket for the client;
- reads are synchronous: reading from an `InputStream` causes the reading process to block; and
- writing is asynchronous: the message is buffered by the `OutputStream`

java.net UDP

- the `DatagramSocket` and `MulticastSocket` classes provide ‘best effort’ communication between two or more processes;
- communication is point-to-point (`DatagramSocket` to `DatagramSocket`) or broadcast (`DatagramSocket` to `MulticastSocket`) and message based;
- naming is indirect and (a)symmetrical—clients name the host and port to which they wish to send a message as part of the datagram, servers accept messages from any (or a specific) client on that port and can also send messages to a group of clients listening on a particular port;
- reads are synchronous: receiving from the socket causes the receiving process to block; and
- writing is asynchronous

Example: KnockKnockServer

The KnockKnockServer tells “knock-knock” jokes:

- uses TCP connections
- the server creates a socket (ServerSocket) on port 4444 and waits for a client to connect
- the socket’s input and output byte streams are wrapped to allow string I/O
- interaction with the user (including the list of jokes) is handled by a separate KnockKnockProtocol class

— *Java Tutorial*

KnockKnockServer 1

```
import java.net.*; import java.io.*;

public class KnockKnockServer {

    public static void main(String[] args) throws IOException {

        // Creating the server socket ...
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(4444);
        } catch (IOException e) {
            System.err.println("Can't listen on port: 4444");
            System.exit(1);
        }
        // continued ...
    }
}
```

Example: KnockKnockSever 2

```
// Creating the client socket ...
Socket clientSocket = null;
try {
    clientSocket = serverSocket.accept();
} catch (IOException e) {
    System.err.println("Accept failed.");
    System.exit(1);
}

// Wrapping the byte streams ...
PrintWriter out = new PrintWriter(
    clientSocket.getOutputStream(),
    true);
```

ServerSocket

- the `accept ()` method waits until a client requests a connection to the host and port of this server
- when the connection is successfully established, the `accept ()` method returns a *new* `Socket` object which is bound to a *new* (dynamically allocated) port
- the server can communicate with the client over this new port while continuing to listen for client connection requests on the `ServerSocket` on the original port
- implies the server is multi-threaded

Example: KnockKnockSever 3

```
// Wrapping the byte streams ...
PrintWriter out = new PrintWriter(
    clientSocket.getOutputStream(),
    true);
BufferedReader in = new BufferedReader(
    new InputStreamReader(
        clientSocket.getInputStream()));

// Reading from and writing to the socket ...
KnockKnockProtocol kkp = new KnockKnockProtocol();
outputLine = kkp.processInput(null);
out.println(outputLine);
```

Example: KnockKnockSever 4

```
// Reading from and writing to the socket ...
KnockKnockProtocol kkp = new KnockKnockProtocol();
outputLine = kkp.processInput(null);
out.println(outputLine);

while ((inputLine = in.readLine()) != null) {
    outputLine = kkp.processInput(inputLine);
    out.println(outputLine);
    if (outputLine.equals("Bye."))
        break;
}

// close streams and sockets and exit ...
}
```

Supporting multiple clients

- the `KnockKnockServer` in the *Java Tutorial* processes a single connection request (client) and then exits
- to allow the server to handle multiple clients, we can create a new thread for each client connection:

```
while (true) {  
    accept a connection ;  
    create a thread to deal with the client ;  
}
```

- the thread reads from and writes to the client connection as necessary

Example: `KKMultiServer`

```
import java.net.*; import java.io.*;  
  
public class KKMultiServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket serverSocket = null;  
  
        try {  
            serverSocket = new ServerSocket(4444);  
        } catch (IOException e) {  
            System.err.println("Can't listen on port: 4444.");  
            System.exit(-1);  
        }  
  
        while (true)  
            Socket socket = serverSocket.accept();  
            new KKMultiServerThread(socket).start();  
    }  
}
```

Example: KKMultiServerThread

```
import java.net.*; import java.io.*;

public class KKMultiServerThread extends Thread {
    private Socket socket = null;

    public KKMultiServerThread(Socket socket) {
        super("KKMultiServerThread");
        this.socket = socket;
    }

    public void run() {
        try {
            PrintWriter out = new PrintWriter(
                socket.getOutputStream(),
                true);
```

© Brian Logan 2007

G52CON Lecture 15: Distributed Processing in
Java

19

Example: KKMultiServerThread 2

```
// The thread's run method ...
public void run() {
    try {
        // Wrapping the byte streams as before ...
        PrintWriter out = new PrintWriter(
            socket.getOutputStream(),
            true);

        BufferedReader in = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));

        // Reading from and writing to the socket as before
        String inputLine, outputLine;

        // omitted ...
    }
}
```

© Brian Logan 2007

G52CON Lecture 15: Distributed Processing in
Java

20

Example: ChatServer

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ChatServer {

    /* The Chatter program    by J M Bishop  January 1997
     * =====
     *                        Java 1.1 January 1998
     * Sets up a chat server for multiple clients.
     * Modified to use java.util linked list instead of
     * Java Gently linked list
     */

    private static LinkedList clientList = new LinkedList();
    private static int id = 0;
```

Example: ChatServer 2

```
public static void main(String[] args) throws IOException {
    // Get the port and create a socket there.
    int port = 8190;
    ServerSocket listener = new ServerSocket(port);

    // Listen for clients. Start a new handler for each.
    // Add each client to the list.
    while (true) {
        Socket client = listener.accept();
        new ChatHandler(client).start();
        System.out.println("New client no. " + id +
            " on client's port " +
            client.getPort());
        clientList.add(client);
        id++;
    }
}
```

Example: ChatServer 3

```
static synchronized void broadcast(String message, String name)
throws IOException {
    // Sends the message to every client including the sender.
    Socket s;
    PrintWriter p;
    for (int i = 0; i < clientList.size(); i++){
        s = (Socket)clientList.get(i);
        p = new PrintWriter(s.getOutputStream(), true);
        p.println(name + ": " + message);
    }
}

static synchronized void remove(Socket s) {
    clientList.remove(s);
    id--;
}
}
```

© Brian Logan 2007

G52CON Lecture 15: Distributed Processing in
Java

23

Example: ChatServer 4

```
class ChatHandler extends Thread {

    /* The Chat Handler class is called from the Chat Server:
    * one thread for each client coming in to chat.
    */

    private BufferedReader in;
    private PrintWriter out;
    private Socket toClient;
    private String name;

    ChatHandler(Socket s) {
        this.toClient = s;
    }

    public void run() {
        try {
```

© Brian Logan 2007

G52CON Lecture 15: Distributed Processing in
Java

24

Example: ChatServer 5

```
public void run() {
    try {
        // Welcome new client to the Chat Room.
        in = new BufferedReader(new InputStreamReader(
            toClient.getInputStream()));
        out = new PrintWriter(toClient.getOutputStream(), true);
        out.println("*** Welcome to the Chatter ***");
        out.println("Type BYE to end");
        out.print("What is your name? ");
        out.flush();
        String name = in.readLine();
        ChatServer.broadcast(name +
            " has joined the discussion.",
            "Chatter");

        // Read lines and send them off for broadcasting.
        while (true) {
```

Example: ChatServer 6

```
        // Read lines and send them off for broadcasting.
        while (true) {
            String s = in.readLine();
            if (s.startsWith("BYE")) {
                ChatServer.broadcast(name +
                    " has left the discussion.",
                    "Chatter");
                break;
            }
            ChatServer.broadcast(s, name);
        }
        ChatServer.remove(toClient);
        toClient.close();
    } catch (Exception e) {
        System.out.println("Chatter error: "+e);
    }
}
```

The next lecture

Distributed processing in Java II

Suggested reading:

- Andrews (2000), chapter 8;
- Java Tutorial Sockets and RMI