Tentamen
Nätverksprogrammering
Lösningsförslag

2009–06–01, 14.00–19.00

---

1. a) Falskt
   b) Sant
   c) Sant
   d) Falskt
   e) Falskt
   f) Sant
   g) Sant
   h) Falskt

2. a) En URL är ett ”namn” på en resurs på Internet (normalt). Den består av tre huvuddelar:
   `protocol://hostname/path/object`
   - Protokolldelen (protocol) som anger accessprotokollet till resursen.
   - Värdmaskinnamnet (hostname) som anger på vilken maskin resursen finns.
   - Sökväg/objektdelen (path/hostname) som anger vilken resurs på värdmaskinen som avses. Denna del kan även kompletteras med en del som anger parametrar till resursen.

   b) Klassen `URL` representerar värdmaskinen för resursen genom att slå upp dess faktiska IP-adress i en DNS-server medan klassen `URI` representerar värdmaskinen i textform. Det senare gör att två URIer kan betraktas som olika då de använder olika aliasnamn trots att de i verkligheten pekar ut samma resurs.

3. *Marshalling* innebär att data som ska skickas mellan klienten och servern vid ett RMI-anrop, dvs metodparametrar och/eller returvärden, konverteras från sin normala datarepresentation till ett externt (serialiserat) format som kan skickas över nätverket.

4. Problemet med att hantera flera olika oberoende öppna nätverksförbindelser med en enda tråd är att vi måste anropa en blockerande läsoperation för att vänta på att något ska anlända på en öppen förbindelse. Medan vi blockerar kan vi inte hantera data som kan tänkas anlända på andra förbindelser. Vi måste ha ett sätt att parallellt bevaka flera öppna förbindelser samtidigt utan att tillgripa busy-wait som pollar förbindelserna hela tiden och därmed slösar bort CPU-tid.

   Lösningen är klassen `Selector` i vilken vi kan registrera ett antal förbindelser (sockets) som vi vill kunna bevaka. När vi sedan anropar `Selector.select()` kommer exekveringen att blockeras tills dess något finns att göra på någon av dessa registrerade förbindelser. När vi kommer ut ur anropet kan vi ur klassen få information om på vilken förbindelse eller vilka förbindelser vi kan läsa/skriva utan att registrera att blockera.

5. The first class of parsers is driven by events in the document, where an event here corresponds to an XML element or an HTML tag. In event-driven parsing, events are mapped onto a set of methods that will be called when they are encountered by the parser. The programmer has to create and associate a class to the parser that contains all or some of the methods and implement them so that they react properly when the parser reads the corresponding element. In the Java HTML parser, the class is ParserCallback. It features six callback methods such as handleStartTag() and handleText(). The callback is associated to the parser using the parse() method.

   The second class of parsers is generic and will load any XML or XHTML document in memory and build a tree out of it. The DOM implementation in Java uses node classes such as Element and Attribute (Attr) to represent elements or attributes in the document. To access, process, or transform the elements, the programmer has to traverse the tree. It can use Java DOM API methods, such as getDocumentElement(), getChildNodes(), getAttributes().

   Event-driven parsing is fast and economical. Tree-model parsing is versatile and generic.

6. Exempel på problem vi måste hantera för strömmande mediaapplikationer:
   - Fördröjningar
   - Jitter
   - Paketförluster

7. a) 1) `wait()` – blockerar den anropande tråden samt låser upp den aktuella monitorns lås så att andra trådar kan gå in i monitorn. Monitorn låses igen innan tråden exekverar vidare efter anropet.
      2) `notify()` – gör att *en* tråd som väntar i ett anrop av `wait()` i den aktuella monitorn, om någon sådan tråd finnes, görs körbar och lämnar anropet av `wait()` så fort som monitorn inte längre är låst.
      3) `notifyAll()` – som `notify()`, fast den gör *alla* väntande trådar körbara.

   b) För att någon de nämnda metoderna ska kunna anropas måste exekveringen befinna sig inne i en metod som är deklarerad `synchronized` i objektet som vi anropar metoden på (eller i ett motsvarande synchronied-block).

8. a) Man måste ”ansluta” socketen till den aktuella multicastadressen. Det gör man lämpligen med ett anrop ”`ms.joinGroup(ia);`ömedelbart före while-satsen. Att multicastadressen sattes i datagramobjektet räcker inte – det behöver man inte göra.

   b) Problemet är att när ett meddelande tas emot sätts den interna längden på datagrampaketet till längden på det mottagna meddelandet. Nästa gång samma datagramobjekt används för att ta emot ett meddelande trunkerar den meddelandet om det är längre än den tidigare längden. Lösningen är antingen att återställa längden på datagrampaketet innan nästa anrop av `ms.receive(dp);` genom att skriva ”`dp.setLength(buf.length);`”, eller att skapa nya datagramobjekt för varje meddelande som tas emot.

1. Filöverföring

Ett vanligt problem med denna uppgift är att designa sitt applikationsprotokoll på ett sådant sätt att mottagaren av data kan avgöra när ett helt kommando eller en hel fil tagits emot över en TCP-förbindelse. När man skriver en följd av strängar/bytes till en socket bildas en kontinuerlig ström av bytes. Det finns inget automatiskt sätt för mottagaren att återskapa de fragment som skrevs till strömmen i andra änden. Applikationen måste själv avgöra var de olika delfälten börjar och slutar. Man kan t.ex. inte avgöra när en sträng har överförts genom att läsa tills metoden `read()` returnerar -1! Detta betyder att strömmen är stängd och att inget mera kan skickas på den någonsin och det är ju inte vad vi är ute efter!

Man kan tänka sig åtminstone två olika alternativa lösningar på uppgiften. Den ena håller en TCP-förbindelse öppen under hela sessionen med användaren. Filnamn som överförs över TCP-förbindelsen avslutas alltid med ett särskilt tecken med koden 0 för att mottagaren ska kunna detektera detta. Filer kan ju innehålla vilka tecken som helst så där fungerar inte den tekniken. I stället skickas först längden på filen i bytes innan själva innehållet skickas. Då vet mottagaren exakt hur många tecken som ska komma.

Lösning nummer två, som redovisas nedan, bygger istället på att en helt ny TCP-uppkoppling sker för varje kommando. När en fil överförts helt stänger sändaren helt enkelt uppkopplingen och filslut kan *i detta fall* detekteras genom att `read()` returnerar -1.

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

public class FileClient {

    public static void main(String args[]) {
        if (args.length!=1) {
            System.err.println("Syntax: java FileClient <target machine>");
        } else {
            try {
                BufferedReader input = new BufferedReader(
                    new InputStreamReader(System.in));
                String command;
                do {
                    System.out.println("Ready!");
                    command = input.readLine();
                    if (command.charAt(0)=='s') {
                        sendFile(args[0],command.substring(2,
                            command.length()));
                    }
                    if (command.charAt(0)=='g') {
                        getFile(args[0],command.substring(2,
                            command.length()));
                    }
                } while (!command.equals("q"));
            } catch (IOException e) {
                System.err.println("An I/O error occurred.");
            }
        }
    }

    private static void sendFile(String target,String name)
            throws IOException {
        FileInputStream f = new FileInputStream(name);
        Socket sock = new Socket(target,3788);
        OutputStream os = sock.getOutputStream();
        os.write('s');
        os.write(name.getBytes());
        os.write(0);
        int ch = f.read();
```

```
        while (ch>=0) {
            os.write(ch);
            ch = f.read();
        }
        f.close();
        sock.close();
    }

    private static void getFile(String target,String name)
            throws IOException {
        FileOutputStream f = new FileOutputStream(name);
        Socket sock = new Socket(target,3788);
        InputStream is = sock.getInputStream();
        OutputStream os = sock.getOutputStream();
        os.write('g');
        os.write(name.getBytes());
        os.write(0);
        os.flush();
        int ch = is.read();
        while (ch>=0) {
            f.write(ch);
            ch = is.read();
        }
        f.close();
        sock.close();
    }
}

----------------

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

class RequestHandler extends Thread {
    private Socket sock;

    public RequestHandler(Socket s) {
        sock = s;
    }

    public void run() {
        InputStream is = null;
        OutputStream os = null;
        int command;

        try {
            is = sock.getInputStream();
            os = sock.getOutputStream();
            command = is.read();
            switch (command) {
                case 'g':
                    transferFile(is,os);
                    break;
                case 's':
                    receiveFile(is);
                    break;
            }
            sock.close();
        } catch(IOException e } {
            try { sock.close(); } catch(IOException e)  {}
        }
    }
```

```
    private void transferFile(InputStream is,OutputStream os)
            throws IOException {
        String filename = readFile(is);
        FileInputStream f = new FileInputStream(filename);
        int ch = f.read();
        while (ch>=0) {
            os.write(ch);
            ch = f.read();
        }
        f.close();
    }

    private void receiveFile(InputStream is) throws IOException {
        String filename = readFile(is);
        FileOutputStream f = new FileOutputStream(filename);
        int ch = is.read();
        while (ch>=0) {
            f.write(ch);
            ch = is.read();
        }
        f.close();
    }

    private String readFile(InputStream is) throws IOException {
        String name = new String();
        int ch = is.read();
        while (ch>0) {
            name = name+((char) ch);
            ch = is.read();
        }
        return name;
    }
}

class FileServer {

    public static void main(String args[]) {
        ServerSocket server = null;
        Socket sock = null;
        try {
            server = new ServerSocket(3788);
            while (true) {
                sock = server.accept();
                new RequestHandler(sock).start();
            }
        } catch(IOException e) {
            System.err.println("An error occurred, terminating...");
            System.exit(1);
        }
    }
}
```

2. Tricket här var att genomskåda att detta inte handlar om något annat än en enkel buffer, eller ”mailbox”, liksom den som implementerades i laboration 2. Vi godkände lösningar med buffertlängden ett (som på laborationen) såväl som lösningar som kunde buffra fler socket-objekt. Nedan visas en enkel lösning med buffertlängden ett.

   Att göra en variant av bufferten som klarar att buffra mer än en begäran är nästan ännu enklare än lösningen nedan om vi förutsätter att vi inte kommer att köra slut på minnet. Då ersätter vi bara attributet request med en lista av något slag och `assignJob()` kommer då aldrig att behöva vänta. Den lägger bara in parametern sist i listan och gör `notify()`. Den andra metoden `awaitJob()` kommer då att vänta tills listan är icke-tom samt då plocka ut det äldsta elementet. Den behöver då *inte* längre heller göra `notifyAll()`.

```
public class ThreadPoolManager {

    private Socket request = null;

    public synchronized void assignJob(Socket s) {
        while (request!=null) {
            try {
                wait();
            } catch(InterruptedException e) {
                // Should not happen
                e.printStackTrace();
                System.exit(1);
            }
        }
        request = s;
        notify();       // notifAll() also OK, but not necessary.
    }

    public synchronized Socket awaitJob() {
        while (request==null) {
            try {
                wait();
            } catch(InterruptedException e) {
                // Should not happen
                e.printStackTrace();
                System.exit(1);
            }
        }
        Socket s = request;
        request = null;
        notifyAll();        // notifAll() necessary in this case.
        return s;
    }
}
```