Tentamen
Nätverksprogrammering
Lösningsförslag

2015–06–04, 14.00–19.00

Del 1

1. 1) Begäran anländer till webbservern.
   2) Om det är första gången som sidan begärs laddas motsvarande ".jsp"-fil in och kompileras om till en servlet av ett särskilt översättningsprogram. I annat fall går vi direkt till steg 4 nedan.
   3) Den nyskapade servleten kompileras av javakompilatorn och startas.
   4) Metoden `service()` anropas av webbservern. Metoden analyserar begäran och skapar HTML-kod som svar.
   5) Den producerade HTML-koden laddas ner till klienten.

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

3. HTML defines a markup language that provides a text document with a structure consisting of sections, character styles, internet links, etc. The markup language has the form of brackets, where each bracket has a type:. The format of an opening bracket is <bracket> with the end bracket being: </bracket>. The bracket are called the elements or tags.

   A HTML document consists of a version information and the root of the document: <html> under which we find the text annotated with the markup elements. An HTML document can be represented as a tree, where the nodes are the HTML elements. Elements may have attributes that provide additional information. The <a> element, describing a string linked to a web page is an example of it that has the href attribute that describes the web page address. HTML parsers can parse documents and produce a parse trees out of them: a hierarchical tree starting at the root element, where the nodes are the HTML elements. HTML parsers can tell if a HTML document is correct and enable programs to traverse, analyze, and transform the tree. The DOM model is a standard representation of HTML documents that enables programs to access the trees. Other parsers can use an event-based mechanism that triggers functions according to the different elements.

   During the lectures and the labs, we have reviewed two tools to parse HTML documents:

   1) one based a built-in java library: HTMLEditorKit, an event-ased parser. It is part of Java, but it has limited parsing capabilities and it is not well documented
   2) one based on an external library: jsoup. It is very complete and easy to use, but it requires an external library.

4. En klient-server-arkitektur för strömmande media består av:

   1) En server som skickar ut ljud eller video med en fördefinierad kodning (t.ex. MPEG) med en viss hastighet (t.ex. 25 bilder/sekund för video) och upplösning (t.ex. 1280x720).
   2) En klient som tar emot mediaströmmen, avkodar den och spelar upp den. För att minska effekten av jitter måste klienten läsa in en mängd data innan uppspelning börjar och lagra den i en buffert.

   Strömmande video kan använda sig av flera olika protokoll:

   1) På nätverksnivå är UDP det snabbaste protokollet, men dess populäritet minskar i Amerika och Europa, dock ej i Kina. Det har inget inbyggt stöd för att garantera att data överförs i lagom hastighet.
   2) TCP blir alltmer populärt då det har inbyggt stöd för felhantering och att anpassa överförings-hastigheten.

   Vi behöver ett video- eller ljudöverföringsprotokoll ovanpå det rena nätverksprotokollet.

   1) RTP är ett standardiserat protokoll som bygger på UDP.
   2) HTTP-baserade protokoll som bygger på TCP är moderna alternativ till UDP då de kan använda konventionella HTTP-servrar. DASH och Apples HTTP live streaming är exempel på två sådana protokoll. Arkitekturen bygger på att servern delar upp mediaströmmen i en mängd filer som överförs sekventiellt m.h.a HTTP-protokollet. Klienten hämtar nästa segment med ett GET-kommando som anger var i strömmen denna vill börja strömma media.

5. Ett kopplat protokoll, såsom TCP, upprättar en (tänkt) fast förbindelse mellan de två kommunicerande noderna över vilken alla meddelanden sedan går. Man behöver alltså inte ange för varje enskilt meddelande vart det ska skickas. Ett okopplat protokoll, såsom UDP, behandlar varje meddelande som helt oberoende av alla andra meddelanden och därför måste destinationsadressen anges för varje meddelande.

6. Busy-wait innebär att en tråd ligger i en loop och väntar på att ett villkor ska bli uppfyllt genom att oavbrutet testa villkoret om och om igen. Detta innebär att den kommer att stjäla CPU-tid från andra trådar som skulle kunna använda tiden till att göra nyttigt arbete och i värsta fall till och med blockra dem helt från att få köra. Ett bättre sätt att vänta på en händelse utan att ta upp värdefull CPU-tid är att använda sig av Javas inbyggda monitorbegrepp och operationerna `wait()` och `notify()`.

7. a) Ömsesidig uteslutning innebär att endast en tråd kan vid varje tillfälle befinna sig inom en och samma s.k. *kritiska region,* dvs exekvera kod som skulle störa exekveringen av en annan tråds kod (t.ex. genom samtidig manipulering av data eller annan resurs). Om en annan tråd försöker gå in i den kritiska regionen måste denna vänta tills den tråd som befinner sig inom regionen lämnar den.

   b) Ändra deklarationerna av `getValue()` och `setValue()` till:

```
public synchronized int getValue() ...

public synchronized void setValue(int newValue) ...
```

   c)
```
public synchronized int getValue() {
   while (value==0) {
      try { wait(); } catch(InterruptedException e) { }
   }
   int v = value;
   value = 0;
   notifyAll();
   return v;
}

public synchronized void setValue(int newValue) {
   while (value!=0) {
      try { wait(); } catch(InterruptedException e) { }
   }
   value = newValue;
   notifyAll();
}
```

8. a) Raden ”`sent = input.read(buffer);`” läser in så många bytes (dock max 128) som finns läsbara från TCP-förbindelsen vid lästillfället (om inga bytes går att läsa blockerar den dock tills minst en byte går att läs). Antalet bytes som läses kan alltså variera från 1 till 128. När vi skriver ut bufferten till `output` skrivs dock alltid 128 bytes. Har mindre lästs in skrivs det alltså ut ett antal extra bytes innehållande skräp på förbindelsen.

   b) Torrent-Tore borde ändra utskriften till ”`output.write(buffer,0,sent);`”. På så sätt skrivs bara de bytes som faktiskt lästes ut.

Del 2

1) a) Apache/2.2.15 (Red Hat)
   b) Se kod nedan.
   c) Se kod nedan.
   d) Se kod nedan.
   e) Se kod nedan.
2) a) $256^2 = 65536$
   b) Om servern svarar precis före timeout i alla lägen kommer det att ta cirka $(200 + 200) \cdot 65536$ millisekunder. Det motsvarar ungefär 7,2 timmar.
   c) Se kod nedan
   d) Se kod nedan

MonoScan

```
public class MonoScan implements Callable<Boolean> {
    private ServerStats serverCnt;
    private String base;

    public MonoScan(String base, ServerStats serverCnt) {
        this.serverCnt = serverCnt;
        this.base = base;
    }

    private String dotAddress2Server(String address) {
        try {
            InetAddress ipAddr = InetAddress.getByName(address);
            URL url = new URL("http://" + ipAddr.getHostName());
            URLConnection uc = url.openConnection();
            uc.setConnectTimeout(200);
            uc.setReadTimeout(200);
            uc.connect(); //This is necessary to start the timeout clock

            String serverType = uc.getHeaderField("Server");
            // Normally, it should call connect(). See JNP 4, page 189.
            // On a Mac, this does not seem to be systematic

            return serverType;
        }  catch (SocketTimeoutException e) {
            return null;
        } catch (IOException e) {
            return null;
        }
    }

    private ServerStats scan() throws IOException {
        for (int i = 0; i < 256; i++) {
            String dotAddress = base + Integer.toString(i);
            String server = dotAddress2Server(dotAddress);
            if (server != null) {
                serverCnt.addServer(server, dotAddress);
            }
        }
        return serverCnt;
    }
```

```
    public static void main(String[] args) throws IOException {
        String base = args[0] + ".";
        ServerStats servers = new ServerStats();
        MonoScan st = new MonoScan(base, servers);
        st.scan();
        servers.print();
    }

    public Boolean call() {
        try {
            scan();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return Boolean.TRUE;
    }
}
```

MultiScan.java

```
public class MultiScan {

    public static void main(String args[]) throws UnknownHostException,
                                InterruptedException, ExecutionException {
        String base = args[0] + ".";
        ServerStats servers = new ServerStats();

        ExecutorService pool = Executors.newFixedThreadPool(32);
        List<Future<Boolean>> futures = new ArrayList<>();
        for (int i = 0; i < 256; i++) {
            String address = base + Integer.toString(i) + ".";
            MonoScan task = new MonoScan(address, servers);
            Future<Boolean> f = pool.submit(task);
            futures.add(f);
        }
        for (Future<Boolean> future : futures) {
            future.get();
        }
        pool.shutdown();

        servers.print();
    }

}
```

ServerStats.java

```
public class ServerStats {
    private Map<String, List<String>> servers;

    public ServerStats() {
        servers = new TreeMap<>();
    }

    public synchronized void addServer(String server, String address) {
        if (servers.get(server) != null) {
            servers.get(server).add(address);
        } else {
            List<String> addressList = new ArrayList<>();
            addressList.add(address);
            servers.put(server, addressList);
        }
    }

    public synchronized void print() {
        for (String server : servers.keySet()) {
            System.out.println(server + ":\t" + servers.get(server).size());
        }
    }
}
```