Tentamen
Nätverksprogrammering
Lösningsförslag

2016–05–31, 8.00–13.00

Del 1

1. a) Vid multicast skickas ett datagram från en sändande dator till flera mottagare samtidigt. Det fungerar genom att de noder som är intresserade av att motta en viss typ av meddelande (meddelande sänt till en utvald s.k. multicast-adress) registrerar detta hos sin router. När ett datagram sänds till den aktuella multicastadressen kommer routrarna i nätverket att se till att meddelandet kommer till rätt mottagare. Den avsändande noden vet alltså inte exakt vilka mottagarna är. Meddelandet kopieras och först när nätverkstopologin så kräver, vilket sparar bandbredd på nätverket.

   b) Varje meddelande förses med en räknare som räknas ned varje gång meddelandet passerar en router. När räknaren blir noll sänds inte meddelandet vidare längre. TTL används för att begränsa spridningen av meddelanden i nätverket.

2. Både servlets och JSP (Java Server Pages) är java-baserade tekniker. JSP är implementerat med hjälp av servlets. När en JSP-sida begärs från en webbserver kommer JSP-sidan automatiskt att kompileras om till en servlet som sedan startas och skapar den efterfrågade webbsidan.

3. a) En frågesträng kan t.ex. innehålla informationen som en användare fyllde i i ett formulär på en webbsida.

   b) Vid ett GET-kommando hakas frågesträngen på URL:en som begärs från webbservern. En praktisk konsekvens av detta är att den är synlig i URL-fältet i en webbläsare. Längden på strängen kan också vara begränsad. Vid POST skickas frågesträngen i stället som meddelandeinnehåll efter de header-rader som följer POST-kommandot i HTTP-protokollet.

4. a) Med java.nio är det möjligt att med en enda tråd bevaka inkommande trafik på flera uppkopplingar utan att använda sig av polling.

   b) I ett `Selector`-objekt kan man registrera ett antal sockets man vill bevaka. Genom att anropa `select()` i objektet kan man sedan blockera tills det går att utföra någon sorts I/O på någon av de bevakade uppkopplingarna.

5. a) 1) `BufferedOutputStream`
      2) `CipherOutputStream`
      3) `FilterOutputStream`
      4) `DataOutputStream`

   b) Den generella skillnaden är att klasserna relaterade till strömmar arbetar med bytes i botten medan Readers/Writers är tecken- eller textorienterade.

6. a) Ibland har man ett större antal oberoende jobb som ska utföras som med fördel kan parallelliseras. Dock vill man ofta undvika att ha för många parallella jobb igång samtidigt. Då passar en trådpool bra. En sådan kan implementeras genom att varje jobb beskrivs av en klass som implementerar `Runnable` (eller `Callable`, om vi vill skapa en s.k. *future*). Vi kan sedan skapa ett objekt av klassen `ExecutorService` och ange hur många jobb som ska kunna vara aktiva samtidigt. Därefter kan vi "submitta" jobb till objektet som ser till att de utförs efter hand.

   b) Ibland vill man att den parallella aktiviteten (jobbet) ska producera något slags resultat som man vill använda senare i en annan tråd. Då kan vi använda oss av en en s.k. *future*. En future är som ett vanligt parallellt jobb fast det finns en funktion man kan anropa för att få tillbaka resultatet av beräkningen. Har beräkningen hunnit avslutas får man svaret direkt, annars blockeras den anropande tråden tills beräkningen är klar.

7. 1) `Falskt`
   2) `Sant`
   3) `Sant`
   4) `Falskt`

8. RTP är ett protokoll avsett för överföring av ljud och bild i realtid, även om protokollet i sig strikt taget inte garanterar realtidsöverföring. Det är ett paketbaserat protokoll som tillhandahåller identifikation av mediatyp, tidsstämplar och sekvensnummer. Det kräver tillägg i form av ett applikationslager för att hantera paket som inte kommer i rätt ordning, jitter och för att kompensera för paketförluster.

   RTCP är en del av RTP-protokollet som används för att förmedla statistik för förbindelsen mellan sändare och mottagare. Exempel på information som överförs är jitterdata, paketförluster och antal sända paket.

   RTSP är ett HTTP-liknande protokoll som används för att kontrollera uppspelningen av strömmande media. Man kan likna det med en fjärrkontroll med funktioner för att starta, stoppa och pausa uppspelningen.

9.

```
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();
}
```

Del 2

Till denna uppgift finns det många alternativ att välja på när man ska designa sin lösning. Man kan använda TCP eller UDP och man kan ha en flertrådad server eller en enkeltrådad. Nedan visas en enkel lösning baserad på TCP och en server med två trådar.

Man kan möjligen invända att med den givna lösnngen finns det en teoretisk chans att det blir hackigt ljud om en klient av någon anledning har en förbindelse som är för långsam. Då kan anropen av `write()` fylla operativsystemets nätverksbuffertar och `write()` blockera. I praktiken såg vi aldrig detta fenomen när vi körde vårt system i början av 90-talet, så jag tycker vi kan bortse från det problemet. Vill man ändå åtgärda det kan man:

- Använda sig av paketet `java.nio` där `Selector`-klassen kan användas för att garantera att ett anrop av `write()` inte blockerar.
- Ha en separat tråd för varje ansluten klient som anropar `write()`. Tråden skulle få sina data från en egen buffert med begränsad storlek som klassen `Transmitter` skulle skriva i. Om bufferten blir full kastar man bort data och bara den långsamma klienten drabbas av störningar.

a) Klientprogram:

```
public class Client {
    public static void main(String [] args]) {
        try {
            SoundIO sound = new SoundIO();
            Socket s = new Socket(args[0],7878);
            InputStream is = s.getInputStream();
            byte [] buffer = new byte[1024];
            int res = is.read(buffer);
            while (res!=-1) {
                sound.write(buffer,res);
                res = is.read(buffer);
            }
            s.close();
        } catch(IOException e) {
            System.err.printn("A network error occured. Aborting...");
        }
    }
}
```

b) Serverprogram:

```
public class Server {
    public static void main(String [] args]) {
        Connections conn = new Connections();
        new Connector(conn).start();
        new Transmitter(conn).start();
    }
}

class Connections{
    private ArrayList<Socket> connected = new ArrayList<Socket>();

    public synchronized void addConnection(Socket s) {
        connected.add(s);
        notify();
    }

    public synchronized void checkConnected() {
        while (connected.size()==0) {
            try { wait(); } catch (InterruptedException e) {}
        }
    }

    public synchronized void sendToAll(byte [] data, int len) {
        ArrayList<Socket> remove = null;
        for(Socket s : connected) {
            try {
                s.getOutputStream().write(data,len);
            } catch(IOException e) {
                if (remove==null) {
                    remove = new ArrayList<Socket>();
                }
                remove.add(s);
            }
        }
        if (remove!=null) {
            for(Socket s : remove) {
                connected.remove(s);
                try { s.close(); } catch(IOException e) {}
            }
        }
    }
}
```

```
class Connector extends Thread {
    private Connections conns;

    public Connector(Connections c) {
        conns = c;
    }

    public void run() {
        try {
            ServerSocket ss = new ServerSocket(7878);
            while (true) {
                Socket s = ss.accept();
                conns.addConnection(s);
            }
        } catch(IOException e) {
            System.err.println("Fatal network error occured. Aborting...");
            System.exit(1);
        }
    }
}


class Transmitter extends Thread {
    private Connections conns;

    public Connector(Connections c) {
        conns = c;
    }

    public void run() {
        try {
            SoundIO sound = new SoundIO();
            byte [] buffer = new byte[1024];

            while(true) {
                conns.checkConnected();
                int res = sound.read(buffer);
                conns.sendToAll(buffer,res);
            }
        } catch(IOException e) {
            System.err.println("Fatal network error occured. Aborting...");
            System.exit(1);
        }
}
```