# Tentamen
# Nätverksprogrammering
# Lösningsförslag

**2008–05–26, 8.00–13.00**

---

1. 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.

2. Båda teknikerna är javabaserade tekniker som används för att skapa dynamiskt webbinnehåll på serversidan, men det som gör att de verkligen är intimt knutna till varandra är att JSP bygger på servlets. En JSP-specifikation kompileras automatiskt om till en servlet vid behov.

3. a) Data som matats in i inmatningsfält i ett HTML-formulär är exempel på information som kan överföras via en frågesträng.

   b) Om GET används överförs frågesträngen som en del av den begärda URL:en. Servern får sedan dela upp URL:en i resursdel och frågesträng. I och med att frågesträngen är en del av URL:en syns den i webbläsarfönstrer, vilket möjligen kan vara oönskat, men det medför också att det är möjligt att göra ett bokmärke som innefattar både adress och frågesträng i webbläsaren. Vid POST sänds frågesträngen som en del av de header-rader som följer på POST-kommandot enligt HTTP-protokollet.

4. a) Det nya paketet `java.nio` har stöd för icke-blockerande I/O och möjlighet att bevaka aktivitet på flera förbindelser parallellt, vilket gör det möjligt att skriva program som hanterar flera parallella förbindelser samtidigt utan att använda sig av trådar.

   b) Klassen `Selector` används för att avgöra på vilka förbindelser det går att utföra I/O just nu eller, om ingen I/O är möjlig för tillfället, vänta på att I/O ska bli möjlig på någon av flera parallella förbindelser.

5. a) Det kommer att verka som om bara kortare och kortare meddelanden, typiskt trunkerade, kan tas emot.

   b) Problemet är att `DatagramPacket`-objektet innehåller ett längd-attribut som dels talar om hur långt ett mottaget meddelande är samt, om objektet används för att ta emot ett meddelande, talar om hur många bytes av meddelande som maximalt ska sparas i bufferten. När vi återanvänder ett `DatagramPacket`-objekt måste vi alltså ställa tillbaka längdattributet så att det blir lika med storleken på den tillgängliga bufferten.

   Ett sätt att lösa problemet är att skapa ett nytt `DatagramPacket`-objekt varje gång vi ska ta emot ett nytt meddelande. Alternativt kan vi återställa längdattributet i det befintliga objektet:

   ```
   dp.setLength(buf.length);
   ```

---

6. Det centrala objektet vid parsning av HTML i Java är ett objekt av typen `javax.swing.html.HTMLEditorKit.Parser`. Detta objekt innehåller en metod `parse` som genomför själva parsningen av sidan. För att skapa objektet finns det en metod i `javax.swing.html.HTMLEditorKit` som heter `getParser`. Tyvärr är denna deklarerad `protected` så för att kunna anropa den är man tvungen att subklassa `javax.swing.html.HTMLEditorKit` och skriva en egen `public`-deklarerad metod som gör anropet.

   Som inparametrar till `parse` anger man dels en `Reader` som är en öppen förbindelse till webbservern/sidan som ska parsas. Denna erhålles m.h.a ett `URL`-objekt som representerar sidan som ska parsas. En andra parameter till `parse` är ett objekt av typen `javax.swing.html.HTMLEditorKit.ParserCallback`. När parsern stöter på olika HTML-element i sidan, t.ex. olika typer av start- och slut-taggar för hyperlänkar o.d. kommer parsern att anropa metoder i `ParserCallback`-objektet som representerar den aktuella taggen. Genom att skapa en subklass till `ParserCallback` och överlagra dessa metoder kan vi styra vad som ska hända när parsern stöter på respektive tagg.

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. Det finns risk för dödläge (deadlock) vid körning av koden. Dödläge kan typiskt uppstå när det förekommer i programmet att trådar låser en resurs, t.ex. en monitor, och sedan försöker låsa ytterligare en resurs. Det krävs då dessutom att den andra resursen redan är låst av en annan tråd som i sin tur därefter försöker låsa den första resursen. Då kommer den första tråden att stå och vänta på att den andra tråden släpper en resurs, men det kan den inte göra eftersom den i sin tur väntar på att den första tråden ska släppa sin resurs. Man kan även tänka sig situationer med fler trådar och längre beroendekedjor inblandade. I det aktuella fallet blir det praktiskt taget garanterat dödläge därför att båda trådarna börjar med att låsa varsin monitor (anropar `m1` i respektive monitor) och sedan, efter en sekunds väntetid och medan monitorn fortfarande är låst, i sin tur låsa den andra monitorn.

10. a) MessageStoreImpl.java

```
import java.util.*;

class MessageStoreImpl implements MessageStore {
    class Message {
        private String receiver;
        private String message;

        public Message(String r, String m) {
            receiver = r;
            message = m;
        }

        public String getReceiver() {
            return receiver;
        }

        public String getMessage() {
            return message;
        }
    }

    private ArrayList<Message> messages = new ArrayList<Message>();

    private Message find(String receiver) {
        for(Message m : messages) {
            if (m.getReceiver().equals(receiver)) {
                return m;
            }
        }
        return null;
    }

    public synchronized void addMessage(String receiver, String message) {
        messages.add(new Message(receiver,message));
        notifyAll();
    }

    public synchronized String nextMessage(String receiver) {
        Message m = find(receiver);
        while (m==null) {
            try {
                wait();
            } catch(InterruptedException e) {}
            m = find(receiver);
        }
        messages.remove(m);
        return m.getMessage();
    }
}
```

b) MessagePassing.java

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

public class MessagePassing {
    public static void main(String[] args) {
        MessageStore messages = new MessageStoreImpl();

        try {
            ServerSocket server = new ServerSocket(12345);
            while(true) {
                Socket s = server.accept();
                new ClientHandler(messages,s).start();
            }
        } catch(IOException e) {
            System.out.println("Fatal I/O error in server thread.");
            System.exit(1);
        }
    }
}
```

ClientHandler.java

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

class ClientHandler extends Thread {
    private MessageStore messages;
    private Socket client;

    public ClientHandler(MessageStore m, Socket c) {
        messages = m;
        client = c;
    }

    public void run() {
        try {
            DataInputStream dis = new DataInputStream(client.getInputStream());
            DataOutputStream dos = new DataOutputStream(client.getOutputStream());

            char command = dis.readChar();
            String receiver = dis.readUTF();
            if (command=='w') {
                String message = dis.readUTF();
                messages.addMessage(receiver, message);
            } else {
                dos.writeUTF(messages.nextMessage(receiver));
            }

        } catch(IOException e) {
            // terminate thread in case of I/O error
        }
        finally {
            try {
                client.close();
            } catch(IOException e) {}
        }
    }
}
```

MessageClient.java

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

public class MessageClient {
    private String server;

    /**
     * Constructor.
     * @param server The network address of the server to connect to.
     */
    public MessageClient(String server) {
        this.server = server;
    }

    /**
     * Send a message to a receiver.
     * @param receiver The receiver of the message.
     * @param message The message to send.
     */
    public void send(String receiver, String message) {
        Socket socket = null;
        try {
            socket = new Socket(server,12345);
            DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
            dos.writeChar('w');
            dos.writeUTF(receiver);
            dos.writeUTF(message);
        } catch(IOException e) {
            // terminate call in case of I/O error
        }
        finally {
            try {
                if (socket!=null) {
                    socket.close();
                }
            } catch(IOException e) {}
        }
    }

    /**
     * Returns the next message for this client. If no message is available,
     * the method blocks until a message arrive.
     * @param receiver The receiver of the message.
     * @returns The message.
     */
    public String receive(String receiver) {
        Socket socket = null;
        String message = null;
        try {
            socket = new Socket(server,12345);
            DataInputStream dis = new DataInputStream(socket.getInputStream());
            DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
            dos.writeChar('r');
            dos.writeUTF(receiver);
            message = dis.readUTF();
        } catch(IOException e) {
            // terminate call in case of I/O error
        }
        finally {
            try {
```

```
                if (socket!=null) {
                    socket.close();
                }
            } catch(IOException e) {}
        }
        return message;
    }
}
```

c) MessageRMIServer.java

```
import java.rmi.*;

interface MessageRMIServer extends Remote {
    void addMessage(String receiver, String message)  throws RemoteException;

    String nextMessage(String receiver) throws RemoteException;
}
```

MessageRMIServerImpl.java

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

class MessageRMIServerImpl extends UnicastRemoteObject implements MessageRMIServer {
    private MessageStore messages;

    public MessageRMIServerImpl() throws RemoteException {
        super();
        messages = new MessageStoreImpl();
    }

    public void addMessage(String receiver, String message) throws RemoteException {
        messages.addMessage(receiver, message);
    }

    public String nextMessage(String receiver) throws RemoteException {
        return messages.nextMessage(receiver);
    }
}
```

MessagePassing.java

```
import java.rmi.*;
import java.net.*;

public class MessagePassing {
    public static void main(String[] args) {
        try {
            MessageRMIServer server = new MessageRMIServerImpl();
            Naming.rebind("MessageServer", server);
        } catch(RemoteException e) {
            e.printStackTrace();
        } catch(MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

MessageClient.java

```
import java.net.*;
import java.rmi.*;

public class MessageClient {
    private MessageRMIServer server;
    /**
     * Constructor.
     * @param server The network address of the server to connect to.
     */
    public MessageClient(String server) {
        try {
            this.server = (MessageRMIServer) Naming.lookup("rmi://"+server+"/MessageServer");
        } catch(RemoteException e) {
        } catch(MalformedURLException e) {
        } catch(NotBoundException e) {
        }
    }

    /**
     * Send a message to a receiver.
     * @param receiver The receiver of the message.
     * @param message The message to send.
     */
    public void send(String receiver, String message) {
        if (server!=null) {
            try {
                server.addMessage(receiver, message);
            } catch(RemoteException e) { }
        }
    }

    /**
     * Returns the next message for this client. If no message is available,
     * the method blocks until a message arrive.
     * @param receiver The receiver of the message.
     * @returns The message.
     */
    public String receive(String receiver) {
        String message = null;
        if (server!=null) {
            try {
                message = server.nextMessage(receiver);
            } catch(RemoteException e) { }
        }
        return message;
    }
}
```