Tentamen
Nätverksprogrammering
Lösningsförslag

2018–03–14, 8.00–13.00

Del 1

1. Applikationslagret levererar data mellan applikationer och till användaren (de tre undre lagren hanterar kommunikation mellan datorer). Exempel på protokoll är HTTP. Transportlagret ansvarar för en pålitlig kommunikation. Protokollet TCP är mest pålitligt: alla paket kommer fram, de kommer fram i samma ordning som de skickades och datan har inte förändrats under överföringen. UDP garanterar bara att datan inte har förändrats (paket kan tappas eller komma fram i annan ordning).

2. Kapplöpning kan uppstå när ett program har flera trådar (parallella aktiviteter). Om programmets korrekthet är beroende på vilken ordning operationer görs och den rätta ordningen inte garanteras uppstår kapplöpning. Lösningen är att synkronisera trådarna för att garantera en korrekt ordning. Detta görs normalt med monitorer. Ett exempel:

```
class Bank{
  private double balance;

  synchronized void deposit(double amount){
    balance = balance + amount;
  }

  synchronized void withdraw(double amount){
    balance = balance - amount;
  }
}
```

   Utan synkronisering kan ett trådbyte ske (och därmed ändring av värdet på balance) efter att balance lästs, men innan det uppdaterade värdet skrivs till attributet.

3. REST api
   a) REST använder HTTP för kommunikation.
   b) REST-operationen anges genom HTTP-protokollet. HTTPs operationerna GET/POST/PUT/-DELETE indikerar läsa/skapa/uppdatera/ta bort.
   c) I svaret på ett HTTP-anrop anges en statuskod. Denna statuskod används i REST för att indikera hurvida REST-operationen lyckades eller inte.
   d) Resurser/objekt identifieras genom en URL i REST-anropet.

4. a) *falskt* - en url får bara innehålla tecken från 7-bitars ASCII, vilket å inte gör. Även mellanslag är förbjudna i url:er.
   b) *falskt* - url är en del mängs av uri.

c) *falskt* - TCP är alltid dubbelriktad, både in- och ut-strömmen på en port är bundna till TCP-protokollet.

d) *falskt* - XML stödjer bara text.

e) *falskt* - normalt kan bara javascript exekveras i webbläsaren.

5. Buffrade strömmar

a) Buffring ökar normalt prestandan då färre paket skickas över nätverket (högre bandbredd, men ökade fördröjningar)

b) Buffring innebär att data lagras i en buffert tills den blir full. Detta skapa problem om applikationen skickar små meddelanden som behöver komma fram snabbt, t.ex. i en chattprogram. Det kan även uppstå problem om det finns en dialog mellan servern och klienten. Senario: en webb-server begär inloggningsuppgifter från webbläsaren som del av ett http-anrop. Om "authenticate" texten fastnar i en buffert på servern kommer klienten aldrig att se det och kommunikationen hänger sig. Lösningen är att använda metoden `flush()` som skickar paket, även om det inte är fullt.

6. a) Lösningen innehåller en *bussy wait*. När bakgrundstråden anropar `pause()` fortsätter den använda CPU-tid för att exekvera while-loopen

b)
```
public class PauseControl {
    private boolean paused = false;

    synchronized public void setPause(boolean state) {
        paused = state;
        notifyAll();
    }

    synchronized public void pause() {
        try {
            while(paused) {
                wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Del 2 – High Score

1.
```
public class HiScoreCommunication {
    static final String url = "scoreserver.megagames.se";
    static final int fetchPort = 9001;
    static final int updatePort = 9000;

    Socket socket;
    DataOutputStream fetcherOS;
    DataInputStream fetcherIS;

    public HiScoreCommunication(){
        try {
            socket = new Socket(url, fetchPort);
            fetcherOS = new DataOutputStream(socket.getOutputStream());
            fetcherIS = new DataInputStream(socket.getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void sendNewScore(HiScore score) throws UnknownHostException, IOException {
        try(Socket sock = new Socket(url, updatePort)) {
            DataOutputStream dos = new DataOutputStream(sock.getOutputStream());
            dos.writeInt(score.score);
            dos.writeUTF(score.name);
            dos.flush();
        }
    }

    public HiScore awaitNewHiScore(int oldScore) throws IOException {
        fetcherOS.writeInt(oldScore);
        fetcherOS.flush();
        int score = fetcherIS.readInt();
        String name = fetcherIS.readUTF();
        return new HiScore(score, name);
    }

    public void close() throws IOException {
        if(socket != null && !socket.isClosed()) {
            socket.close();
        }
    }
}

public class ServerMain {
    static final int fetchPort = 9001;

    public static void main(String[] args) {
        ScoreMonitor mon = new ScoreMonitor();
        new ScoreUpdateThread(mon).start();
        try(ServerSocket serverSocket = new ServerSocket(fetchPort)){
            while(true) {
                Socket socket = serverSocket.accept();
                new ScoreServiceThread(mon, socket).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
class ScoreMonitor{
    private int score = 0;
    private String name = "no high score set";

    synchronized public void registerNewScore(int newScore, String newName) {
        if(newScore > score) {
            score = newScore;
            name = newName;
            notifyAll();
        }
    }

    synchronized public HiScore awaitNewHiScore(int oldScore){
        try {
            while( oldScore >= score ) {
                wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return new HiScore(score, name);
    }

}

class ScoreServiceThread extends Thread{
    DataInputStream dis;
    DataOutputStream dos;
    private ScoreMonitor mon;

    ScoreServiceThread(ScoreMonitor mon, Socket socket) throws IOException{
        this.mon = mon;
        dis = new DataInputStream(socket.getInputStream());
        dos = new DataOutputStream(socket.getOutputStream());
    }

    public void run() {
        try {
            while(true) {
                int oldScore = dis.readInt();
                HiScore score = mon.awaitNewHiScore(oldScore);
                dos.writeInt(score.score);
                dos.writeUTF(score.name);
                dos.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ScoreUpdateThread extends Thread{
    static final int updatePort = 9000;
    private ScoreMonitor mon;

    ScoreUpdateThread(ScoreMonitor mon){
        this.mon = mon;
    }

    public void run() {
        try(ServerSocket serverSocket = new ServerSocket(updatePort)){
```

```
            while(true) {
                Socket socket = serverSocket.accept();
                DataInputStream dis = new DataInputStream(socket.getInputStream());
                int newScore = dis.readInt();
                String name = dis.readUTF();
                socket.close();
                mon.registerNewScore(newScore, name);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```