Tentamen
Nätverksprogrammering
Lösningsförslag

2018–04–11, 8.00–13.00

Del 1

1. a) IP-numret identifierar en enskild dator på internet.
   b) Port-numret anger vilken tjänst/program som paketen ska skickas till.
   c) DNS är en tjänst som omvandla namn till IP-nummer, t.ex. `cs.lth.se` → 130.235.97.220. Port-numret fås antingen genom defaulvärde för protokollet, `http` → 80, eller så anges det explicit i url:en `http://cs.lth.se:8080/edaf65`.

2. Trådar läggs i tillståndet *waiting* när de anropar `sleep(time)`. De flyttas sedan till tillståndet *ready* efter en viss tidpunkt. Trådar läggs i tillståndet *blocked* när de tvingas vänta på att komma in i monitorn vid anropar til en monitor-metod, d.v.s. en metod som är `synchronized`. Detta inträffar när monitorn ägs av en annan tråd.

3. a) Kaka lagrar text, strängar. Max storlek på en kaka är 4KB. Applikationen bestämmer själva semantiken för texten.
   b) En kaka är kopplad till en domän, t.ex. `cs.lth.se`. När en http-request kommer från den domänen skickar http-serven automatiskt kakan till webb-läsaren som ett http-header-attribut i svaret.

4. a) Varje tecken har en unik unicode *key point*, d.v.s. det är ett värde som identifierar ett tecken. För "basic lattin" är *key point* samma som ascii-koden (`0x00` – `0x7F`).
   b) UTF-8 variabel länd kodning. Varje tecken är 1-4 bytes. Tecknen med *key point* `U-0000` – `U-007F` lagras i en byte. Tecknen med *key point* `U-0080` – `U-07FF` lagras i två bytes. Den första byten inleds med bitarna `110`, följt av 5 bitar från *key point* värdet. Byte två inleds med `10` följt av 6 bitar från *key point* värdet. På samma sätt används tre bytes för *key point* värden `U-0800` – `U-FFFF` (`1110xxxx 10xxxxxx 10xxxxxx`) och fyra bytes för resten (`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`.

5. a) *falskt* – UDP garanterar inte att paketen kommer fram, men innehållet i paket som gör det är korrekt.
   b) *sannt* – I ett XML-dokument finns bara ett element på översta nivån.
   c) *falskt* – Man måste normalt även använda `wait()` och `notifyAll()` för att undvika *busy wait*.
   d) *sannt* – Både XML och JSON är textbaserade format som kan användas för att skicka information mellan en server och en klient.
   e) *falskt* – TCP-paket kan skickas till port alla portar.
   f) *falskt* – En css-fil kan inte innehålla JavaScript.

(3p)

6. Betrakta JavaScript-koden

a) Operationen `===` utför ingen typkonvertering, d.v.s. returnerar true on både värde och typ är samma. Operationen `==` utför typkonvertering innan värdet jämförs. `3.14 === '3.14'` är falskt medans `3.14 == '3.14'` är sant.

b) *Block scope* (`let presentOne`) innebär att variabeln är synlig inom det blocket den är deklarerad (samma som i Java). *Function scope* (`var presentTwo`) innebär att variabeln är synlig i hela funktionen.

(2p)

c) Document Object Model(DOM) är ett träd som representerar hemsidan (html och css). Det är DOMen som webbläsaren renderar.

7. a) Om det sker både en insättning och uttag på ett konto samtidigt uppstår kapplöpning mellan get/set-anropen från `Communication`.

b)
```
public class Account {
    private String accountNbr;
    private double balance;

    synchronized public void withdraw(double amount) {
        this.balance = balance - amount;
    }

    synchronized public double deposit(double amount) {
        this.balance = balance + amount;
    }

    /**
     * More code that is not relevant for this question
     **/
}



public class Communication extends thread {
    private Bank bank;

    public Communication(Bank bank){
        this.bank = bank;
    }

    public void run(){
        // The code for communication with ATM, mobile app and webb
    }

    private void withdraw(double amount, String accountNbr){
        Account account = bank.findAccount(accountNbr);
        account.withdraw(amount);
    }

    private void deposit(double amount, String accountNbr){
        Account account = bank.findAccount(accountNbr);
        account.withdraw(amount);
    }
}
```

Del 2 – Budgivning

1. Att lämna bud och att lystna efter nya bud görs på olika portar.

lämna bud

Klienten kopplar upp sig och skickar objektId följt av amount i det format som Javas DataOutputStream använder för `int`. Serven svarar sedan med true/false i det format som Javas DataOutputStream använder för `boolean`. Serven stänger uppkopplingen efter att ha skickat sitt svar.

lyssna efter nya bud

Klienten kopplar upp sig mot serven. Serven håller kopplingen uppe och skickar alla accepterade bud till klienten, samma format som ovan (objektId följt av amount i det format som Javas DataOutputStream använder för `int`). Serven svarar sedan med true/false i det format som Javas DataOutputStream använder för `boolean`. Serven stänger uppkopplingen efter att ha skickat sitt svar.

implementering

```
public class Bid {
  static final int bidDistributionPort = 9001;
  static final int makeBidPort = 9000;

  public int objectId;
  public int amount;

  Bid(){}

  Bid(int objectId, int amount){
    this.objectId = objectId;
    this.amount = amount;
  }

  static Bid read(DataInputStream dis) throws IOException{
    Bid result = new Bid();
    result.objectId = dis.readInt();
    result.amount = dis.readInt();
    return result;
  }

  public void write(DataOutputStream dos) throws IOException {
    dos.writeInt(objectId);
    dos.writeInt(amount);
    dos.flush();
  }

  int getObjectId(){
    return objectId;
  }

  int getAmount(){
    return amount;
  }
}
```

```
public class AuctionCommunication {
    static final String url = "auction.com";

    Socket socket;
    DataInputStream fetcherIS;

    public AuctionCommunication(){
        try {
            socket = new Socket(url, Bid.bidDistributionPort);
            fetcherIS = new DataInputStream(socket.getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public boolean sendNewBid(Bid bid) throws UnknownHostException, IOException {
        try(Socket sock = new Socket(url, Bid.makeBidPort)) {
            DataOutputStream dos = new DataOutputStream(sock.getOutputStream());
            DataInputStream dis = new DataInputStream(sock.getInputStream());
            bid.write(dos);
            return dis.readBoolean();
        }
    }

    public Bid awaitNewBid() throws IOException {
        return Bid.read(fetcherIS);
    }

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


public class ServerMain {
  public static void main(String[] args) {
    new ServerMain();
  }

  public ServerMain() {
    // monitors
    BidFIFO fifo = new BidFIFO();
    ClientList clients = new ClientList();

    // threads for distributing bids
    new BidListenerManager(clients).start();
    new BidDistributor(fifo, clients).start();

    // code for receiving new bids
    try(ServerSocket serverSocket = new ServerSocket(Bid.makeBidPort)){
      while(true) {
        Socket socket = serverSocket.accept();
        new BidReceiver(fifo, socket).start();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
/**
 * Queue bids for distributed to all clients.
 * When a new bid is offered, this monitor checks if it is accepted.
 */
class BidFIFO {
  Queue<Bid> fifo = new LinkedList<Bid>();
  Auctioneer auctioneer = new Auctioneer();

  synchronized public boolean offerNewBid(Bid bid) {
    boolean accepted = auctioneer.makeBid(bid);
    if(accepted) {
      fifo.add(bid);
      notifyAll();
    }
    return accepted;
  }

  synchronized public Bid awaitNewBid() {
    try {
      while( fifo.isEmpty() ) {
        wait();
      }
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return fifo.remove();
  }
}


/**
 * Receive a new bid from a client.
 * Manage the network communication.
 * Closes the socket after replying to the client.
 */
class BidReceiver extends Thread {

  private BidFIFO mon;
  private Socket socket;

  BidReceiver(BidFIFO mon, Socket socket) throws IOException {
    this.mon = mon;
    this.socket = socket;
  }

  public void run() {
    try {
      DataInputStream dis = new DataInputStream(socket.getInputStream());
      DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
      Bid bid = Bid.read(dis);
      boolean accepted = mon.offerNewBid(bid);
      dos.writeBoolean(accepted);
      socket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
/**
 * Keeps track of all connected clients.
 * Sends new bids to all clients
 */
class ClientList {
  ArrayList<DataOutputStream> clients = new ArrayList<DataOutputStream>();

  synchronized public void addClient(DataOutputStream dos) {
    clients.add(dos);
  }

  synchronized public void distributeNewBid(Bid bid) {
    ArrayList<DataOutputStream> closedConnections = new ArrayList<DataOutputStream>();
    for(DataOutputStream dos: clients) {
      try {
        bid.write(dos);
      } catch (IOException e) {
        closedConnections.add(dos);
      }
    }
    for(DataOutputStream dos: closedConnections) {
      clients.remove(dos);
    }
  }
}


/**
 * Accepts new clients for bid distribution
 */
class BidListenerManager extends Thread {
  private ClientList clients;

  BidListenerManager(ClientList clients) {
    this.clients = clients;
  }

  public void run() {
    try(ServerSocket serverSocket = new ServerSocket(Bid.bidDistributionPort)) {
      while(true) {
        Socket socket = serverSocket.accept();
        clients.addClient(new DataOutputStream(socket.getOutputStream()));
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

```
/**
 * Send the bids to all known clients
 */
class BidDistributor extends Thread {
  private BidFIFO bidQueue;
  private ClientList clients;

  BidDistributor(BidFIFO bidQueue, ClientList clients) {
    this.bidQueue = bidQueue;
    this.clients = clients;
  }

  public void run() {
    while(true) {
      Bid bid = bidQueue.awaitNewBid();
      clients.distributeNewBid(bid);
    }
  }
}
```