Föreläsning 9

- Client/Server - Serversidan
- TCP/IP (och UDP)

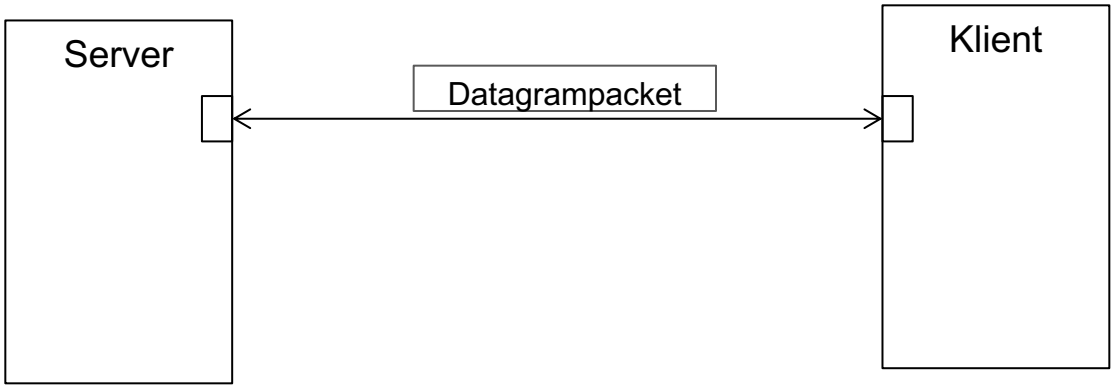
JNP: s 283 - 309

Client/Server, UDP

Klassen ***DatagramSocket*** används på både serversidan och klientsidan.

Servern lyssnar efter *request* på en speciell port och ger *response* på klientens port.

All kommunikation förpackas i ***DatagramPacket***-objekt.



Client/Server, TCP

I java används klassen ***ServerSocket*** för anslutning av klienter.

Vid anslutning av en klient skapas en Socket för kommunikation med klienten.

Kommunikationen sker via en ***InputStream*** och en ***OutputStream*** i vardera enhet

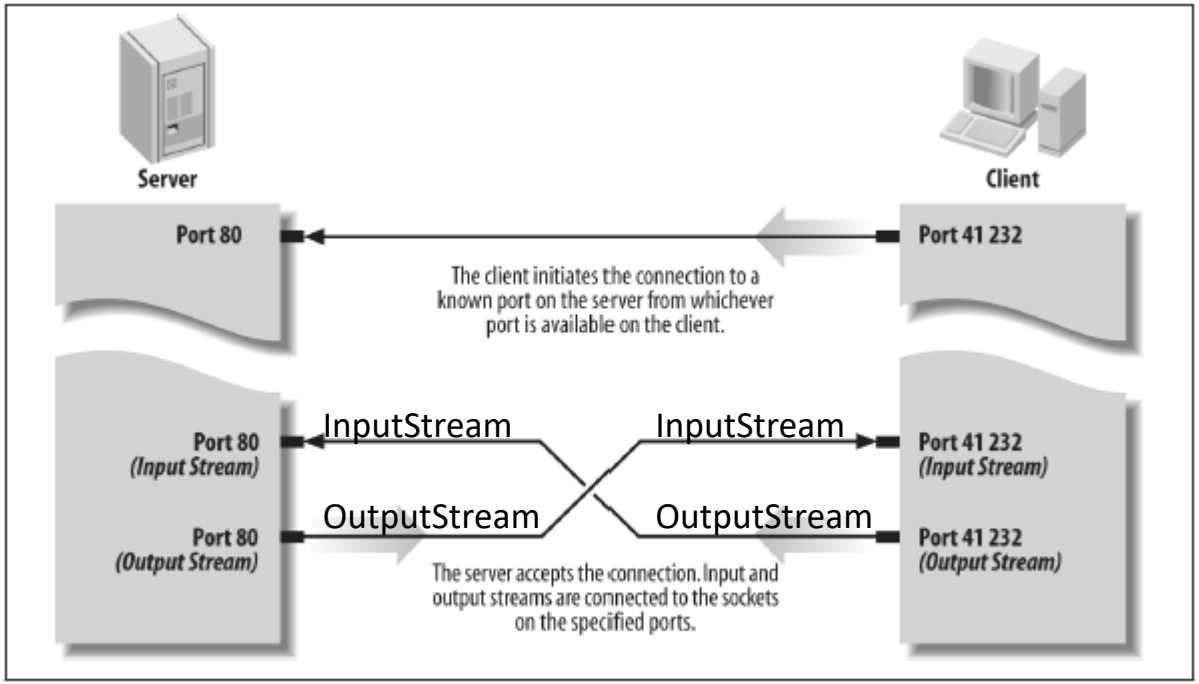


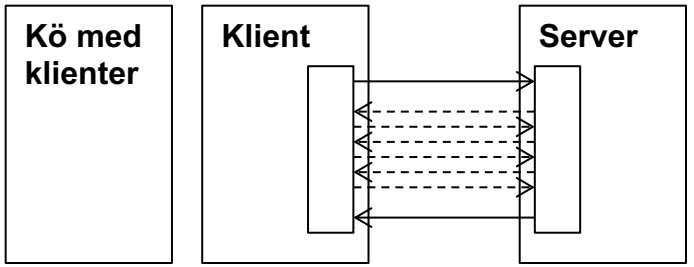
Figure 1-5. A client/server connection

Olika typer av servrar

Iterativ server

En klient i taget hanteras av servern

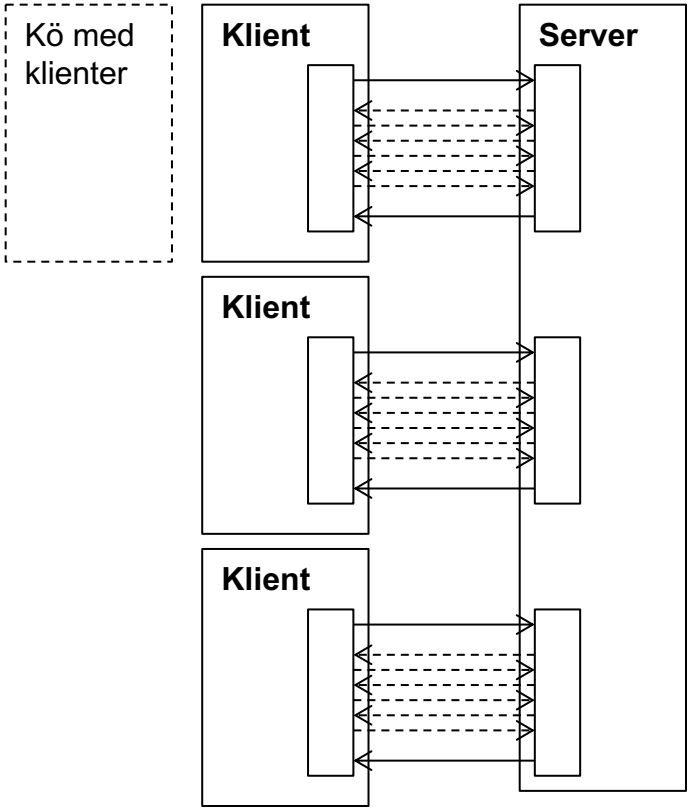
- Då en request exekveras snabbt
- Enklare att konstruera servern



Flertrådad server

Flera klienter hanteras av servern samtidigt

- Då en request tar tid att exekvera
- Mer komplex konstruktion av servern



Iterativ server

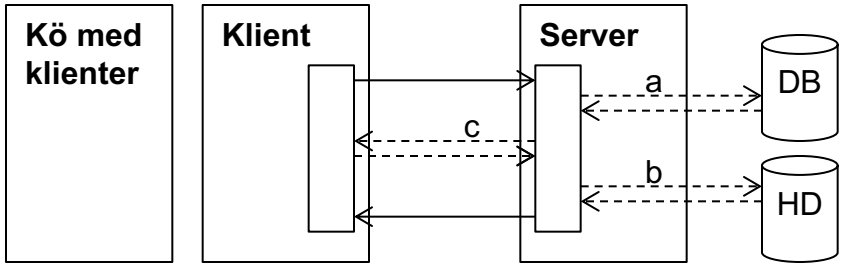
En iterativ server hanterar en klient i taget.

Fördelar

- Effektiv då overhead med trådhantering försvinner
- Enkel att skriva

Nackdelar

- Om en request tar tid att exekvera så får klienter i kö vänta på sin service. Om en operation medför att servertråden får vänta/blockeras viss tid så drabbas klienterna i kön. Detta kan ske t.ex. vid kommunikation med en databas (a), en hårddisk (b) eller klienten (c).
- Om servertråden inte får en resurs den väntar på kan servern helt blockeras eller bli betydligt långsammare.



Iterativ server, UDP

En iterativ server som använder UDP är enkel att skriva.

1. Skapa DatagramSocket och lite till.
2. Lyssna efter request
3. Hantera request
4. Skicka response
5. Tillbaka till steg 2

```
public Server(port) throws SocketException {
    socket = new DatagramSocket(port);
    thread.start();
}

public void run() {
    DatagramPacket packet;
    String response, ticket;
    byte[] buffer = new byte[256];
    byte[] outData;
    while(true) {
        try {
            packet = new DatagramPacket(buffer,buffer.length);
            socket.receive(packet);

            ticket = new String(packet.getData(),0,packet.getLength());
            response = checkTicket(ticket);

            outData = response.getBytes();
            packet = new DatagramPacket(outData,outData.length,
packet.getAddress(),packet.getPort());
            socket.send(packet);
        } catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

I LotteryServerA är *checkTicket* långsam (0-3 sek).

LotteryServerA.java

LotteryClientAC.java

Iterativ server, TCP

En iterativ server som använder TCP är enkel att skriva.

1. Skapa ServerSocket och lite till.
2. Lyssna efter anslutande klient
3. Skapa strömmar
4. Läs request
5. Hantera request
6. Skicka response
7. Stäng anslutning
8. Tillbaka till steg 2

```
public Server(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    thread.start();
}

public void run() {
    String ticket,response;
    System.out.println("Server running");
    while(true) {
        try (Socket socket = serverSocket.accept();
             DataInputStream dis =
                 new DataInputStream( socket.getInputStream());
             DataOutputStream dos =
                 new DataOutputStream( socket.getOutputStream())) {
            ticket = dis.readUTF();

            response = checkTicket(ticket);

            dos.writeUTF(response);
            dos.flush();
        } catch(IOException e) {
            System.err.println(e);
        }
    }
}
```

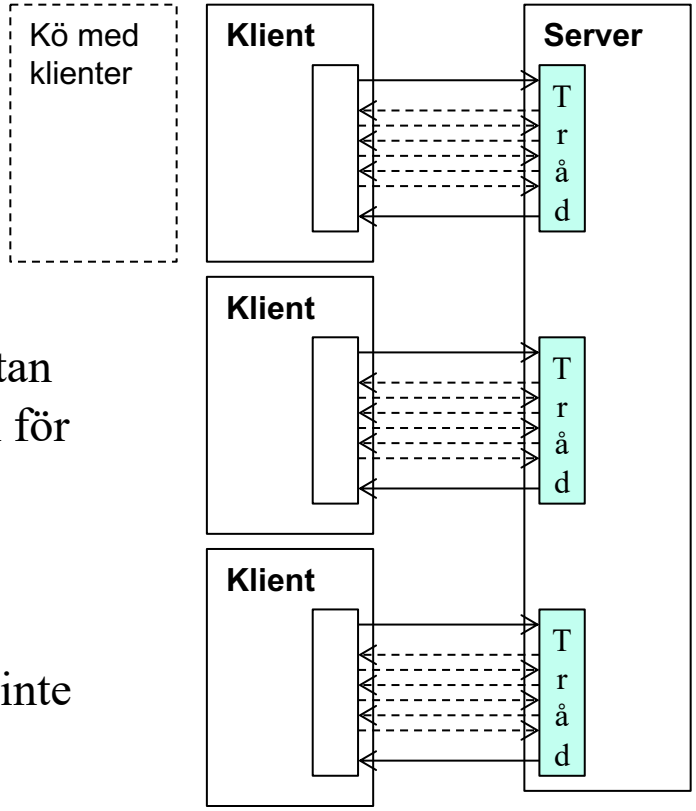
I LotteryServerB är *checkTicket* långsam (0-3 sek).

LotteryServerB.java

LotteryClientBDF.java

Flertrådad server

En flertrådad server hanterar ett antal klienter samtidigt.



Fördelar

- Om tråden, vilken hanterar request från en klient, tillfälligt avbryts i väntan på en resurs påverkas inte prestandan för andra klienter

Nackdelar

- Vid exekvering på en processor ökar inte den totala prestandan i servern.
- Mer komplex att skriva. Gemensamma resurser måste synkroniseras.

Flertrådad server

En tråd per klient

- Server startar en ny tråd för varje ny klient.
- Om klienterna är många samtidigt kan systemet tyngas ner.
- En del resurser går åt till trådhanteringen

Tråd-pool

- Servern startar ett visst antal trådar vilka servar klienter som kopplar upp
- Om klienterna är många så hinner kanske inte trådarna med. Det innebär att klienten får vänta på sin tur eller kanske t.o.m. inte får någon anslutning

Flertrådad server, UDP

En design för en flertrådad server är att låta varje klient behandlas av en tråd, här instans av *ClientHandler*.

1. Skapa DatagramSocket och lite till.
2. Lyssna efter request
3. Skapa tråd som hanterar request
4. Tillbaka till steg 2

```
public Server(int port) throws SocketException {
    socket = new DatagramSocket(port);
    thread.start();
}

public void run() {
    DatagramPacket packet;
    byte[] buffer = new byte[256];
    String ticket;
    System.out.println("Server running");
    while(true) {
        try {
            packet = new DatagramPacket(buffer,buffer.length);
            socket.receive(packet);

            ticket = new String(packet.getData(),0,packet.getLength());
            new ClientHandler(packet.getAddress(), packet.getPort(), ticket);

        } catch(Exception e) {
            System.err.println(e);
        }
    }
}
```

LotteryServerC.java

LotteryClientAC.java

Flertrådad server, UDP

En design för en flertrådad server är att låta varje klient behandlas av en tråd, här instans av *ClientHandler*.

1. Tråd hanterar request, skickar response och avslutas

```
private class ClientHandler extends Thread {
        private InetAddress address;
        private int port;
        private String ticket;

        public ClientHandler(InetAddress address, int port, String ticket) {
            this.address = address;
            this.port = port;
            this.ticket = ticket;
            start(); // tråden startar sig själv
        }

        public void run() {
            try {
                String response = getResponse(ticket);
                byte[] outData = response.getBytes();
                DatagramPacket packet = new DatagramPacket(outData,
outData.length, address, port);
                socket.send(packet);
            } catch (IOException e) {}
        }
    }
```

I LotteryServerC är *getResponse* långsam (0-3 sek).

LotteryServerC.java

LotteryClientAC.java

Flertrådad server, TCP

En design för en flertrådad server är att låta varje klient behandlas av en tråd, här instans av *ClientHandler*.

1. Skapa ServerSocket och lite till.
2. Lyssna efter anslutande klient
3. Skapa tråd som hanterar request och starta tråden.
4. Tillbaka till steg 2

```
public LotteryServerD(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    server.start();
}

public void run() {
    System.out.println("Server running");
    while(true) {
        try  {
            Socket socket = serverSocket.accept();
            new ClientHandler(socket).start();
        } catch(IOException e) {
            System.err.println(e);
        }
    }
}
```

LotteryServerD.java

LotteryClientBDF.java

Flertrådad server, TCP

En design för en flertrådad server är att låta varje klient behandlas av en tråd, här instans av *ClientHandler*.

1. Tråd hanterar request, skickar response och avslutas

```
private class ClientHandler extends Thread {
      private Socket socket;

      public ClientHandler(Socket socket) {
         this.socket = socket;
      }

      public void run() {
         String ticket,response;
         try (DataOutputStream dos =
                  new DataOutputStream( socket.getOutputStream());
               DataInputStream dis =
                  new DataInputStream( socket.getInputStream())    ) {
            ticket = dis.readUTF();
            response = getResponse(ticket);
            dos.writeUTF(response);
            dos.flush();
         } catch(IOException e) {}
         try {
            socket.close();
         } catch(Exception e) {}
      }
   }
```

I LotteryServerD är *getResponse* långsam (0-3 sek)

LotteryServerD.java

LotteryClientBDF.java

Flera request i samma uppkoppling

En server kan tillåta att en klient behåller uppkopplingen under flera request. Det är klienthanteraren som förändras.

I while-loopen läser och skriver tråden upprepade gånger.

Då klienten stänger sin socket kastas IOException (dis.readUTF, dos.writeUTF eller dos.flush). Undantaget fångas efter while-loopen varvid klienttråden avslutas.

```
private class ClientHandler extends Thread {
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        String ticket,response;
        System.out.println("Klient uppkopplad");
        try (DataOutputStream dos =
                new DataOutputStream( socket.getOutputStream());
             DataInputStream dis =
                new DataInputStream( socket.getInputStream())    ) {
            while(true) {
                ticket = dis.readUTF();
                response = getResponse(ticket);
                dos.writeUTF(response);
                dos.flush();
            }
        } catch(IOException e) {}
        try {
            socket.close();
        } catch(Exception e) {}
        System.out.println("Klient nerkopplad");
    }
}
```

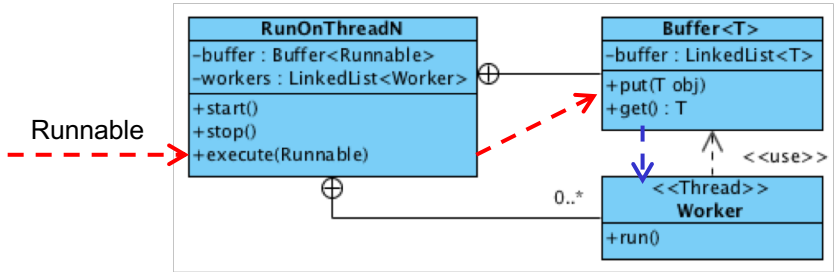
- LotteryServerE.java
- LotteryClientE.java
- UI.java

Trådpool hanterar klienter

En trådpool är ett antal trådar vilka utför uppgifter vilka lagras i en buffert.

Om varje ansluten klient placeras i bufferten kan trådarna successivt beta av klienterna.

RunOnThread (F7) går ganska enkelt att göra om till en trådpool vilken exekverar Runnable-implementeringar.



Runnable-implementeringar placeras i bufferten genom anrop till metoden *execute(Runnable r)*.

Lediga trådar ligger och väntar på att en *Runnable*-implementering placeras i bufferten. När så sker hämtas implementeringen av en av de lediga trådarna och metoden *run()* exekveras.

```
public void run() {
   while(!Thread.interrupted()) {
      try {
         buffer.get().run();
      } catch(InterruptedException e) {
         break;
      }
   }
}
```

RunOnThreadN

Trådpool hanterar klienter

I RunOnThread behövs det en buffert och en collection för att hålla reda på trådarna:

```
private Buffer<Runnable> buffer = new Buffer<Runnable>();
private LinkedList<Worker> workers;
```

Trådarna måste instantieras och startas. Instansvariabeln *n* håller antalet trådar.

```
public synchronized void start() {
   Worker worker;
   if(workers==null) {
      workers = new LinkedList<Worker>();
      for(int i=0; i<n; i++) {
         worker = new Worker();
         worker.start();
         workers.add(worker);
      }
   }
}
```

Trådarna bör kunna avslutas – Detta kan ske omedelbart (eventuella objekt i bufferten exekveras ej) eller då bufferten är tömd (se alternativ stop-metod).

```
public synchronized void stop() {
   if(workers!=null) {
      buffer.clear();
      for(Worker worker : workers) {
         worker.interrupt();
      }
      workers = null;
   }
}
```

RunOnTreadN

Trådpool hanterar klienter

Skillnaderna mot LotteryServerD är ganska små:

- En trådpool instansieras och startas i konstruktorn.

```
public LotteryServerF(int port, int nbrOfThreads) throws IOException {
   pool = new RunOnThreadN(nbrOfThreads);
   serverSocket = new ServerSocket(port);
   pool.start();
   server.start();
}
```

- Den inre klassen *ClientHandler* är ingen tråd utan implementerar *Runnable*. En tråd i trådpoolen exekverar run-metoden.

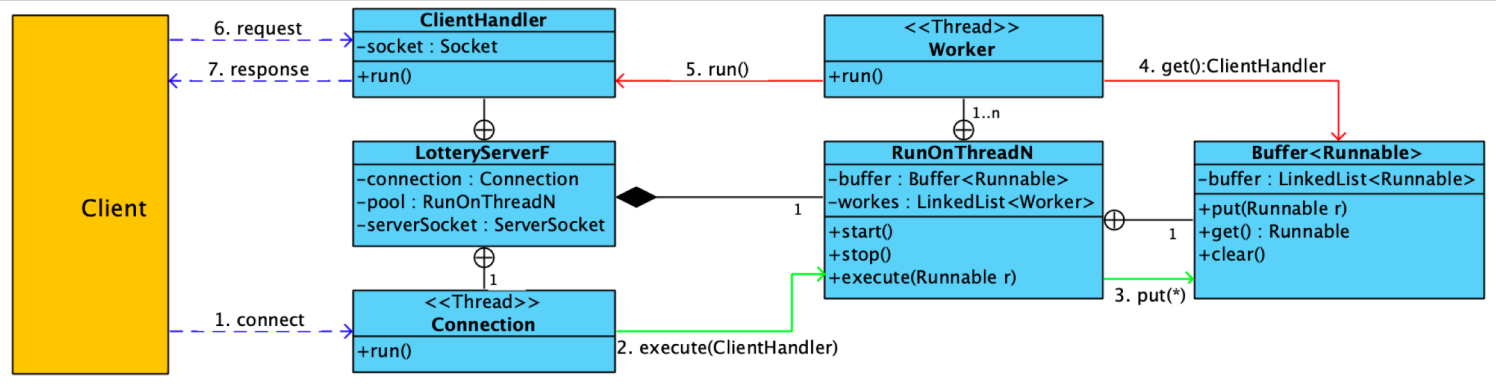
```
private class ClientHandler implements Runnable {
   public void run() {
      // kod som exekveras av tråd
   }
}
```

- Efter att en klient anslutit placeras klienthanteraren i trådpoolens buffert för att exekveras när tid finns.

```
public void run() {
   System.out.println("ServerF running, port: " + serverSocket.getLocalPort());
   while(true) {
      try  {
         Socket socket = serverSocket.accept();
         pool.execute(new ClientHandler(socket));
      } catch(IOException e) {}
   }
}
```

LotteryServerF.java

Trådpool hanterar klienter



1. En klient kopplar upp sig. Det är *run*-metoden i *Connection*-instansen som lyssnar efter klienter som vill koppla upp sig.
2. En instans av *ClientHandler* skapas och *execute(ClientHandler)* anropas.
3. *RunOnThread* placerar *ClientHandler*-instansen i buffertengenom anropet *put(ClientHandler)*.
4. Alla *Worker*-instanser, som ej exekverar en *run*-metod, anropar *get()* i bufferten
5. En Worker-instans får som retur på 4. en *ClientHandler*-instans *och* anropar *run*-metoden i *ClientHandler*-instansen.
6. Klienten skriver en sträng till servern, vilken hanteras av *:LotteryServerF*.
7. Servern svarar med en sträng till klienten (*:LotteryServerF -> :ClientHandler ->* klient)

LotteryServerF.java

LotteryClientBDF.

Server notifierar klienter

En server kan notifiera en klient vid en speciell tidpunkt.

UDP

Servern måste lagra klientinfo (InetAddress/ip-adress + port) i någon form av collection.
Vid notifieringen skickas DatagramPacket till klienten.

TCP

Klienten ska ha en ServerSocket vilken ligger och lyssnar på en speciell port.
Servern måste lagra klientinfo(InetAddress/ip-adress + port på vilken klienten lyssnar efter uppkoppling) i någon form av collection.
Vid notifiering kopplar servern upp mot klienten, överför data och kopplar ner.

AlarmServer.java

AlarmClient.java

Loggning i servern

Det är ofta lämpligt att logga aktiviteter och fel som inträffar i servern. Klassen ***Logger*** innehåller stöd för trådsäker loggning.

- Få tillgång till en eller flera instanser av klassen *Logger*:

```
public class LotteryServerD implements Runnable{
    private final static Logger requestLog = Logger.getLogger(”requests");
    private final static Logger errorLog = Logger.getLogger("errors");
```

- Skapa en ***FileHandler*** för loggning till fil

```
    private FileHandler requestFile = new FileHandler(”files/requestLog.log");
    private FileHandler errorFile = new FileHandler("files/errorLog.log");
```

Nedanstående inställningar görs i konstruktorn:

- Ändra eventuellt loggningens format (xml är default vid loggning i fil)

```
    requestFile.setFormatter(new SimpleFormatter());
```

- Koppla FileHandler-instansen till Logger-instansen

```
    requestLog.addHandler(requestFile); // log to file
    errorLog.addHandler(errorFile); // log to file
```

- Loggningen sker default i Console-fönstret. Om loggningen inte ska ske i Console-fönster så stäng av den:

```
    requestLog.setUseParentHandlers(false); // no log to console
```

Loggning kan ske på olika nivåer, t.ex. INFO, WARNING, SEVERE och det finns metoder att anropa med Logger-instansen

```
requestLog.info(socket.getInetAddress() + ": Lott=" +ticket+", …"+response);
errorLog.warning(e.toString());  // e – Exception-instans
errorLog.severe(e.toString());  // e – Exception-instans
```

LotteryServerD.java