NTNU
Kunnskap for en bedre verden

# *Sockets*

*Tomas Holt, Institutt for datateknologi og informatikk ved NTNU*
*30.09.2015*
*Lærestoffet er utviklet for faget IFUD1042 Applikasjonsutvikling for mobile enheter*

*Resymé: Denne leksjonen tar for seg socket-programmering. Dette er lavnivå nettverksprogrammering som gir mye kontroll og store muligheter.*

# Contents



# 6 Sockets

## 6.1 Om leksjonen

Vi skal i denne leksjonen se mer på nettverksprogrammering via socket-programmering. Vi går da på nivået under HTTP. Dette er ikke tatt opp i boka så leksjonen og nettressurser må brukes.

Når det gjelder socketprogrammering er erfaringen at det kan være litt vanskelig (spesielt når man skal gjøre dette med emulator). Det er derfor lurt **å gjøre ting så enkelt** som mulig før man prøver å lage mer kompliserte løsninger. Vedlagt leksjonen finner du

eksempelkode av enkleste sort og vil sammen med beskrivelsen i leksjonen være et godt utgangspunkt til å komme i gang med denne typen nettverksprogrammering.

Merk at det i leksjonen brukes en del kommandoer fra konsoll. For at dette skal fungere må man ha man ha satt miljøvariablen PATH på maskina. Denne må inneholde <androidsdk>/tools og <androidsdk>/platform-tools, hvor <androidsdk> må endres til katalogen hvor du installerte Android SDK. Hvordan sette PATH ble tatt opp i boka i forbindelse med installasjon av Android Studio/Android SDK, men sammen med denne leksjonen ligger også et skriv som forklarer hvordan man bruker miljøvariabler (som f.eks. PATH) for de som evt. trenger dette.

## 6.2 Sockets i stedet for HTTP

Vi har tidligere kikket ganske grundig på bruk av HTTP-protokollen for å kunne kommunisere med web-tjenere. HTTP har imidlertid en svakhet som vi så langt ikke har omtalt, den er nemlig kun beregnet for miljøer der det er et klart skille mellom tjener og klient. Klienten gjør en forespørsel og tjeneren gir respons. Hva om vi ønsker oss et system hvor tjeneren kan ta kontakt med klienten (da er egentlig rollene snudd om, slik at klienten plutselig er tjener og motsatt)? Med HTTP er ikke dette mulig.

En socket er enkelt og greit definert som et endepunkt i en nettverksforbindelse. Man har dermed to sockets i en forbindelse. En socket er da unikt bestemt av sin ip-adresse og portnummer. Portnummer brukes for å kunne ha flere tjenere og klienter på samme maskin. Vi kan f.eks. ha en web-tjener på port 80 og vår egenlagde tjener på port 81. En HTTP-forbindelse består altså av et socketpar, så hva er nytt? HTTP er en protokoll som bestemmer hvordan klient- og tjener-applikasjoner skal oppføre seg i forbindelse med datautvekslinger. Denne begrensningen slipper vi unna om vi bruker socket-programmering, vi kan dermed selv bestemme hvordan dataene skal organiseres (applikasjonslagprotokollen).

Sockets er mye brukt til nettverksprogrammering, og kan i prinsippet brukes med ulike nettverksprotokoller. Når det gjelder Java og Android så snakker vi om UDP, TCP eller Bluetooth. TCP og Bluetooth er en forbindelsesorienterte protokoller som betyr at man setter opp en forbindelse, og man har dermed garanti for at det man sender over forbindelsen enten kommer fram eller at man får beskjed om at det er skjedd noe feil. Når det gjelder UDP så er dette en forbindelsesløs protokoll. I dette tilfellet sender man bare datapakker ut på nettverket, men man har i dette tilfellet ingen garantier (så lenge ikke mottaker sender en datapakke tilbake - men dette må da i så fall programmeres selv). Du kan selv lese mer om UDP, TCP og porter på [Network basics].

Vanligvis brukes forbindelsesorienterte protokoller (som også ligger til grunn for HTTP via TCP), så i etterfølgende omtales sockets over TCP. Ved å bruke klassene java.net.ServerSocket

og Socket (brukes for TCP) kan vi sette opp en nettverksforbindelse mellom to applikasjoner. Forbindelsen kan holdes åpen så lenge som ønskelig og vil være slik at applikasjonene kan kommunisere begge veier. Et eksempel på hvor slik toveis kommunikasjon er nødvendig kan være en chat, hvor to (eller kanskje også flere) snakker med hverandre over et nettverk. La oss se på på et veldig enkelt eksempel, der poenget er å sette opp en nettverksforbindelse og sende tekst i mellom to android-applikasjoner (du kan også finne mye av den samme informasjonen på [Java Socket]).

Merk at man må bruke ulike klasser for socket-kommunikasjon avhengig av om man velger UDP (DatagramSocket), TCP (ServerSocket og Socket) eller Bluetooth (BluetoothServerSocket og BluetoothSocket).

### 6.2.1 Tjener

Vi starter med å lage tjeneren. Denne startes først og vil stå å vente på oppkobling fra klient(er). Det er altså viktig i hvilken rekkefølge dette skjer. I dette eksemplet har jeg skilt ut nettverkskoden i en egen klasse, og koden vil kjøre i en egen tråd. La oss først se på aktiviteten for tjeneren.

```
1  public class ServerActivity extends Activity {
2
3      @Override
4      public void onCreate(Bundle savedInstanceState) {
5          super.onCreate(savedInstanceState);
6          setContentView(R.layout.main);
7          new Server().start();//start server
8      }
9  }
```

Som du kan se så gjøres det ikke mye. Det eneste interessante er egentlig den siste linjen som oppretter et trådobjekt av klassen Server og starter en ny tråd. Klassen Server ser slik ut:

```
1  public class Server extends Thread{
2          private final static String TAG = "ServerThread";
3          private final static int PORT = 12345;
4
5          public void run() {
6                  ServerSocket ss         = null;
7                  Socket s                        = null;
8                  PrintWriter out         = null;
9                  BufferedReader in       = null;
10
11                 try{
12                         Log.i(TAG,"start server....");
13                         ss = new ServerSocket(PORT);
14                         Log.i(TAG,"serversocket created, wait for client....");
15                         s = ss.accept();
16                         Log.v(TAG, "client connected...");
17                         out = new PrintWriter(s.getOutputStream(), true);
18                         in = new BufferedReader(
19                                 new InputStreamReader(s.getInputStream()));
```

```
20
21                          out.println("Welcome client...");//send text to client
22
23                          String res = in.readLine();//receive text from client
24                          Log.i(TAG,"Message from client: " + res);
25          } catch (IOException e) {
26                  e.printStackTrace();
27          }finally{//close sockets!!
28                          try{
29                          out.close();
30                                  in.close();
31                          s.close();
32                          ss.close();
33                  }catch(Exception e){}
34          }
35          }
36  }
```

Koden ss = new ServerSocket(PORT); sørger for å "å lage en tjener-socket". Ved å kalle accept() på denne så vil socketen stå å lytte etter klientoppkoblinger. Når en klient kobler til vil det returneres **en ny socket** (her kalt s) som kan brukes til å kommunisere med klienten. Dette gjøres for å fortsatt kunne bruke ServerSocket'en til å lytte etter klienter (må da kalle accept() på nytt igjen - noe som igjen vil returnere en ny socket når en ny klient kobler til).

Når man først har en socket-oppkobling mot klienten så bruker man socket-objektet til å hente java-strømmer for å kommunisere med klienten (via s.getInputStream() og s.getOutputStream()). For å gjøre lesing og skriving til disse strømmene så enkel som mulig pakker vi dem inn i PrintWriter og BufferedReader-objekter.

Vi har altså mulighet til toveis kommunikasjon, og bruker strømmene in og out for å hhv. lese og skrive mot klienten. Man er imidlertid på ingen måte låst til å bruke de strømklassene som vist i eksemplet. Man kan f.eks. sende egne objekter over slike forbindelser via ObjectOutputStream og ObjectInputStream. Kravet er da at objektene må være serialiserbare. Det vil si at de må implementere interface'et java.io.Serializable.

En ting som er verdt å merke seg er at noen av metodene er blokkerende. readLine()-metoden som brukes vil f.eks. blokkere helt til en hel linje med tekst er mottatt.

Strømmer er mye brukt i Java. De brukes i forbindelse med socket-programmering, men også ved filbehandling. Det er mye informasjon å finne om temaet og for de interesserte kan man finne det meste på [Java IO].

### 6.2.2 Klient

Når vi skal lage klienten trenger vi å vite hvor tjeneren befinner seg. Vi må altså ha maskinnavn/IP-adresse, samt portnummer (12345 i tjenerkoden vår). **Merk at litt spesielle forutsetninger gjelder om du bruke socket-forbindelser i emulator/AVD** (se da eget kapittel om dette). La oss se på hvordan klientkoden blir (utelater aktiviteten foreløpig).

```
1  package leksjon.socket;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.PrintWriter;
7  import java.net.Socket;
8  import android.util.Log;
9
10 public class Client extends Thread {
11         private final static String TAG = "Client";
12         private final static String IP = "10.0.2.2";
13         private final static int PORT = 12345;
14
15         public void run() {
16                 Socket s                = null;
17                 PrintWriter out         = null;
18                 BufferedReader in       = null;
19
20                 try {
21                     s = new Socket(IP, PORT);
22                     Log.v(TAG, "C: Connected to server" + s.toString());
23                     out = new PrintWriter(s.getOutputStream(), true);
24                     in = new BufferedReader(new InputStreamReader(s.getInputStream()));
25
26                     String res = in.readLine();
27                     Log.i(TAG,res);
28                     out.println("PING to server from client");
29                 }catch (IOException e) {
30                     e.printStackTrace();
31                 }finally{//close socket!!
32                         try{
33                                 out.close();
34                                 in.close();
35                                 s.close();
36                         }catch(IOException e){}
37                 }
38         }
39 }
```

I dette tilfellet opprettes socket-forbindelsen ved å gjøre s = new Socket(IP, PORT); Det er alt som skal til så lenge IP-adressen og portnummeret som brukes er gyldig. Dette vil gi klienten en socket-forbindelse mot tjeneren. Når vi først har socket-objektet tilgjengelig kan vi hente ut strømmene vi trenger for å kommunisere med tjeneren.

En ting du godt kan merke deg i koden er rekkefølgen man leser og skriver på. Klienten starter med å lese via in.readLine(). Ettersom klienten nå er blokkert til det kommer noe tekst er det viktig at tjeneren starter med å sende tekst (via out.println()). Hvis både klient og tjener skulle kalle readLine() først ville begge bli hengende å vente på tekst i all evighet.

For å kompletere eksemplet viser jeg under aktiviteten for klientsiden:

```
1  package leksjon.socket;
2
3  import android.app.Activity;
4  import android.os.Bundle;
```

```
5
6  public class ClientActivity extends Activity {
7      /** Called when the activity is first created. */
8      @Override
9      public void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.main);
12         new Client().start();//start client
13     }
14 }
```

### 6.2.3 Tjener med flere klienter

Om man ønsker en tjener som støtter flere klienter kan tjenerkoden være slik:

```
ServerSocket ss = new ServerSocket(PORT);
while (true){
        Socket s = ss.accept();
        new ClientHandlerThread(s).start();
}
```

Som du kan se av koden så vil tjeneren bare gå i løkke og motta klientoppkoblinger. Hver gang en ny klient kobler til, så opprettes en ny tråd av typen ClientHandlerThread, som så startes. Denne tråden vil håndtere all kommunikasjon mot klienten. Tjeneren går så tilbake til å lytte etter nye klientoppkoblinger.

ClientHandlerThread er et trådobjekt som vi selv må lage. I konstruktoren tar vi inn socketen som gjør at vi i objektet får tilgang til nødvendige strømmer. I run()-metoden brukes så strømmene til å kommunisere med klienten (se kap. ”Tjener”).

## 6.3 Socket-forbindelser i emulator (AVD)

Når man skal bruke socket-forbindelser i emulator er det litt spesielle forutsetninger man skal være klar over. Skal man bruke både tjener og klient(er) via emulator så må det opprettes en emulator for hver av dem. I eksemplet for denne leksjonen så trenger vi altså to emulatorer (og ettersom all utskrift skjer til log trenger vi også en logg for hver av dem - se neste kapittel). Lag en som du kaller klient og en som du kaller tjener (det gjør det enkelt å skille dem). Disse kan du opprette i Android Studio (du gjorde dette i starten av faget) eller du kan gjøre det direkte via kommandolinja (forutsetter at PATH er satt som beskrevet i kap. 1.1) slik:

```
android create avd --name tjener --target android-22
android create avd --name klient --target android-22
```

Over opprettes to emulatorer for Android versjon 5.1 (alias android-22).

Neste skritt er å sette opprette ett Android Studio-prosjekt for tjeneren og ett for klienten (kjent stoff). Hvert av disse prosjektene må så konfigureres slik at man ved kjøring velger

riktig emulator. Klient-prosjektet må altså velge emulatoren ”klient” og tjener-prosjektet emulatoren ”tjener”.

Start så de to emulatorene, via et av alternativene

1. finn fram emulatoren (AVD) i Android Studio og velg start
2. start de to prosjektene (noe som også starter emulatoren)
3. kommandoene:
    - emulator -avd tjener
    - emulator -avd klient

Hvis vi nå tar en titt på tittelen i vinduet der emulatoren kjører, så vil vi se noe slikt: "5554:tjener" og "5556:klient". Dette kan bekreftes ved hjelp av adb kommandoen:

```
adb devices
```

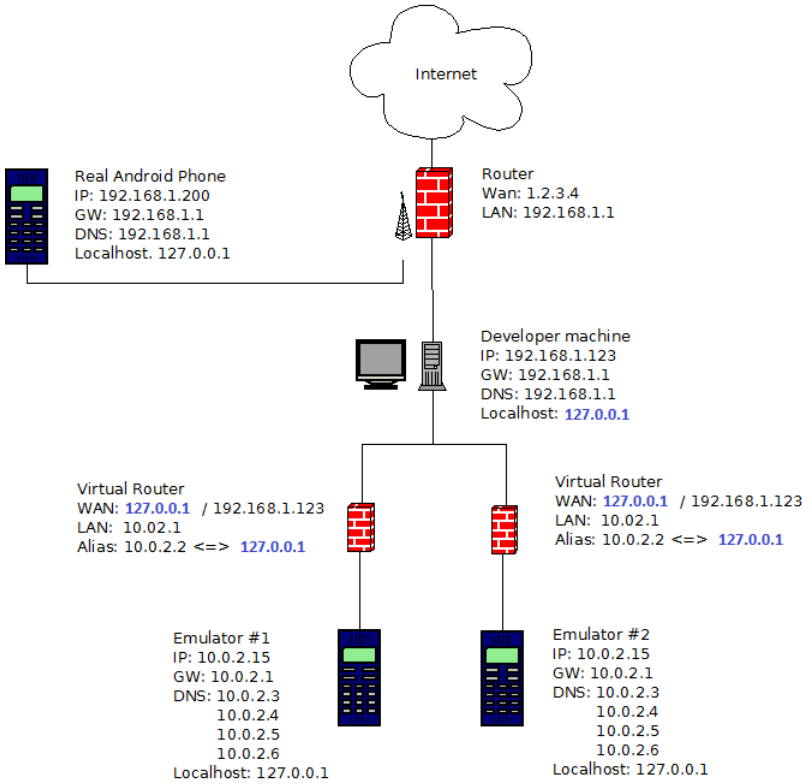
Du vil da se noe slikt:

```
List of devices attached
emulator-5556 device
emulator-5554 device
```

Du kan her se at identifikatoren for ”klientemulatoren” er emulator-5556 og ”tjeneremulatoren” er emulator-5554. Dette må vi nå bruke. Det er slik at hver emulator ligger bak en virtuell ruter (se figur under). **For at klienten skal kunne finne tjeneren i dette tilfellet må vi i klientkoden bruke IP-adressen 10.0.2.2** (dette er gjort i eksempelkoden). Dette gjør at forbindelsen settes opp mot vår PC, men denne forbindelsen må så rutes til riktig emulator (tjeneremulatoren). Dette gjøres via kommandoen

```
adb -s emulator-5554 forward tcp:12345 tcp:12345
```

Dette sørger nå for å redirigere all TCP-trafikk på port 12345 til emulator-5554 (på TCP port 12345). Altså mottar tjeneren nettverkstrafikken vi ønsker.




## 6.4 LogCat og emulator

Når man kjører flere emulatorer samtidig så er det greit å kunne koble LogCat til emulatorene. Om emulatoren stater fra Android Studio så vil du se loggen i LogCat-vinduet. Om emulatoren startes fra konsoll så kan du få tilgang til loggen via denne kommandoen:

```
adb -s emulator-5556 logcat
```

All utskrift til loggen vil nå vises i konsollet hvor du skrev kommandoen. I vårt eksempel vi det være naturlig å gjøre det samme også for emulator-5554 (i et annet konsollvindu).

For mer generell informasjon om bruk av kommandoer fra kommandolinja se [Android Tools]. Spesielt interessante er [Adb] og [Emulator] som også er brukt som utgangspunkt for kommandoene i leksjonen.

## 6.5 Referanser

[Adb] - http://developer.android.com/guide/developing/tools/adb.html

[Android Tools] - http://developer.android.com/guide/developing/tools/index.html

[Emulator] - http://developer.android.com/guide/developing/tools/emulator.html

[Java IO] - http://download.oracle.com/javase/tutorial/essential/io/

[Java Socket] - http://download.oracle.com/javase/tutorial/networking/sockets/

[Network basics] - http://download.oracle.com/javase/tutorial/networking/overview/networking.html