Lab 7

[Obligatorisk] Multipla klienter

Denna laboration syftar till att ge studenten fördjupad kunskap inom socketprogrammering. Mer specifikt skall man här implementera en chat-server som kan hantera *flera samtidiga* klienter.

7.1 Problemet

I denna laboration skall man implementera en *Chat-Server* som kan hantera flera samtidiga klienter. Då man skriver ett sådant serverprogram måste man lösa följande problem:

> **Problemet:** Då ett serverprogram läser data med `sock.recv()` *blockeras* programmet tills klienten skickar data, och under tiden som servern är blockerad kan den inte hantera andra klienter!

Man kan lösa detta problem på minst tre olika sätt:

1. Man kan låta server *spinna en ny exekveringstråd* varje gång en ny klient ansluter, så att man hanterar varje klient i en oberoende exekveringstråd. Denna lösning har dock minst två olika nackdelar:
   (a) Om trådarna behöver accessa gemensamt data finns det risk att en tråd ändrar i detta data samtidigt som en annan tråd läser data. Resultatet kan bli fullständigt oförutsägbart. Man kan lösa detta problem, men programmet kan då bli onödigt komplicerat och om man programmerar oskickligt kan det också uppstå risk för *deadlock*.
   (b) Att använda en tråd/klient löser faktiskt inte hela problemet i den aktuella tillämpningen! Under tiden som en servertråd

väntar på svar från sin klient (med det blockerande anropet `sock.recv()`) kan det nämligen hända att servern måste skicka data till just denna klient...

2. Man kan använda en *asynkron* variant av socket-API. Den asynkrona varianten av `recv` kommer inte att blockera och vänta på klientens data. Istället ser den till så att en s.k. *call-back* -funktion blir anropad då klientens data så småningom kommer. Eftersom den s.k. *call-back* funktionen normalt anropas i en annan tråd måste man fortfarande hantera gemensamt data på lämpligt sätt.

3. I denna laboration skall vi använda en tredje metod: Vi skall skriva en enkeltrådad server. Istället för att ligga och vänta på en viss klient skall programmet ligga och vänta på *alla* klienter samtidigt. Så fort någon av dessa klienter skickar data, eller om en ny klient vill ansluta sig, kommer programmet att vakna upp för att behandla denna klient.

7.2 Så skall servern fungera

Uppgiften är att med hjälp av lite skelettkod skriva en mycket enkel variant av en *chat-server*. Servern skall ha följande egenskaper:

1. Servern skall vara enkeltrådad, skriven i python, och använda sig av *select* för att hantera multipla klienter.

2. Flera (obegränsat antal) klienter skall kunna ansluta sig.

3. Så fort en ny klient ansluter sig skall serven skicka meddelandet:

   [100.101.102.103:9090] (Uppkopplad)

   till *samtliga* klienter. (*100.101.102.103:9090* skall bytas mot den aktuella klientens ip-adress och portnummer).

4. Så fort en klient skickar ett meddelande (data) till servern skall servern skicka detta meddelande till samtliga anslutna klienter (inklusive till den klient som skickade meddelandet), så här:

   [100.101.102.103:9090] $M\ e\ d\ e\ l\ a\ n\ d\ e\ t$

5. Så fort en klient kopplar ned sin förbindelse skall servern skicka meddelandet:

   [100.101.102.103:9090] (Nedkopplad)

till alla andra klienter.

Figur 7.2 på sidan 5 visar tre olika *klienter* som anslutit sig till servern och skickat in var sitt meddelande till denna.

7.3 Uppgift / Skelettkod

Skelettkoden i figur 7.1 är halvfärdig. Använd *inte* copy n' paste utan *knappa* in den, förstå den, och ersätt `pass` och `# TODO` med lämplig kod!

Programmet använder en socket för varje klient, och därtill ytterligare en socket (`sockL`) för att lyssna efter nya klienter. Variabeln `listOfSockets` är en lista som innehåller alla dessa socket-objekt.

Anropet

```
tup = select.select( listOfSockets, [], [] )
```

gör så att servern stannar upp och blockerar tills någon socket i listan har inkommande data (eller någon klient kopplar ned). Det är alltså denna kodrad som får servern att vänta på *alla* klienter *samtidigt*, och dessutom samtidigt vänta på sin egen lyssnar-socket (`sockL`). Läs mer om anropet till *select* i dokumentationen[1].

Testa!

Testa servern med hjälp av lämpligt klientprogram! Exempelvis *telnet*, *netcat*, eller programmet *klient.exe* som ligger i pingpong. Det senare programmet är ett Windows-form program som jag hämtat från en nätartikel om socketprogrammering. Figur 7.2 visar programmet då det körs.
Koppla upp minst tre klienter till din server (be gärna en klasskamrat koppla upp sig fån sin maskin). Knappa in några meddelanden från varje klient och verifiera att servern beter sig enligt anvisningarna.

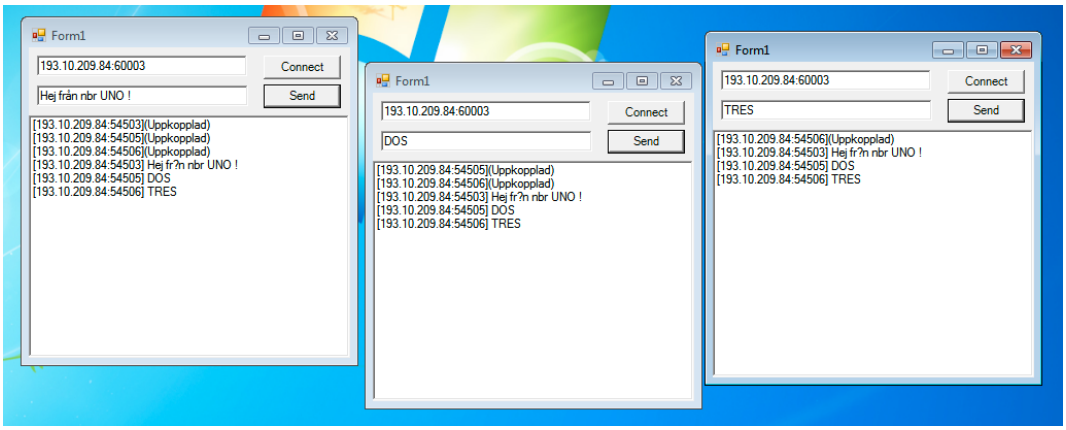
[1] https://docs.python.org/3/library/select.html

```
import socket
import select

port = 60003
sockL = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sockL.bind(("", port))
sockL.listen(2)

listOfSockets = [sockL]

print("Lyssnar på port {}".format(port))

while True:
    tup = select.select(listOfSockets, [], [])
    sock = tup[0][0]

    if sock==sockL:
        pass
        # TODO Ny klient ansluter sig.
        # anropa (sockClient, addr) = sockL.accept() och ta hand om den nya klienten
        # stoppa in klientens socket i listOfSockets
    else:
        # Befintlig klient skickar data eller kopplar ned...
        data = sock.recv(2048)
        if not data:
            pass
            # TODO hantera neddoppning från sock
            # Stäng socketförbindelsen och ta bort sock från listan
        else:
            pass
            # data är ett meddelande från klienten
            # skicka detta till alla klienter
```

Figur 7.1: Skelettprogram.

7.4 Redovisa

Ladda upp det färdiga programmet till pingpong och redovisa sedan muntligt.



Figur 7.2: Tre klienter har anslutit sig och skickat var sitt meddelande. Klientprogrammet av John McTainsh har hämtats från *http://www.codeproject.com/Articles/1608/Asynchronous-socket-communication*