DATANET

Weekly assignment 3

Martin Jørgensen (tzk173)
martinnjorgensen@gmail.com

May 21, 2013

1 Theoretical part

1.1 The Internet Protocol

1.1.1 Addresses and network masks

Part1:

1. The purpose of network masking is to allow the network/internet to identify several machines behind a single router using as few addresses as possible. It also allows for seperation of networks into different blocks (subnets.)
2. `255.225.255.0` is a valid subnet mask since *binary-and* can still be applied over the mask. `/28` should be interpreted as the 28 most significant bits being one, giving a mask of: `11111111.11111111.11111111.11110000` = `255.255.255.240` giving 16 subnets with 14 hosts each.
3. Classfull *CIDR* networks allows for only 3 subnet masks:
   - Class A: 8-bit subnet mask or (255.0.0.0)
   - Class b: 16-bit subnet mask or (255.255.0.0)
   - Class c: 24-bit subnet mask or (255.255.255.0)

   Classless allows subnet masks from 0-bit to 32-bit. 32 and 31 bit masks being useless though.
4. The subnet mask itself tells us the IP configuration of the network, number of possible subnets, sizes etc. Using the the subnet mask on any obtained IP's from the network then allows for the computation of an actual internal IP, which can be used to see what subnets are "populated".

Part2:

1.
   - Subnet Size: 30 hosts
   - Network address: `130.225.165.16`
   - Mask: `255.255.255.224`
   - Broadcast address: `130.225.165.31`
   - 1st IP: `130.225.165.1`
   - 5th IP: `130.225.165.5`
   - Last IP: `130.225.165.30`
2.
   - Subnet Size: 254 hosts
   - Network address: `10.0.42.0`
   - Mask: `255.255.254.0`
   - Broadcast address: `255.255.254.255`
   - 1st IP: `255.255.254.1`
   - 5th IP: `255.255.254.5`
   - Last IP: `255.255.254.254`
3.
   - Subnet Size: 1 host
   - Network address: `4.2.2.1`

- Mask: `255.255.255.255`
- Broadcast address: N/A only one host.
- 1st IP: `4.2.2.1`
- 5th IP: N/A
- Last IP: N/A

4.
- Subnet Size: 16382 hosts
- Network address: `192.38.64.0`
- Mask: `255.255.192.0`
- Broadcast address: `192.38.127.255`
- 1st IP: `192.38.64.1`
- 5th IP: `192.38.64.5`
- Last IP: `192.38.127.254`

1.1.2 Network Address Translation

Part 1:
The NAT maps all internal addresses(Ip,port) via a translation table so it can manage the relationships between different internal and external connections. It maps both IP and port on both sides, allowing different processes from same IP to work as long as the NAT have available ports.
Part 2:
Since a NAT requires memory for the table racking up a lot of long term connections can fill up it's memory. another factor is the fact that the port field is a fixed 16-bits meaning all the ports can be taken up, there is 65535 ports available.
Part 3:
The NAT breaks the layering because it modifies the TCP/UDP headers ,thereby breaking into the 4th layer (transport).

1.2 Distributed Hash Tables

1. **TODO!**
2. **TODO!**
3. **TODO!**

1.3 SSH Tunneling

1.3.1 Part 1

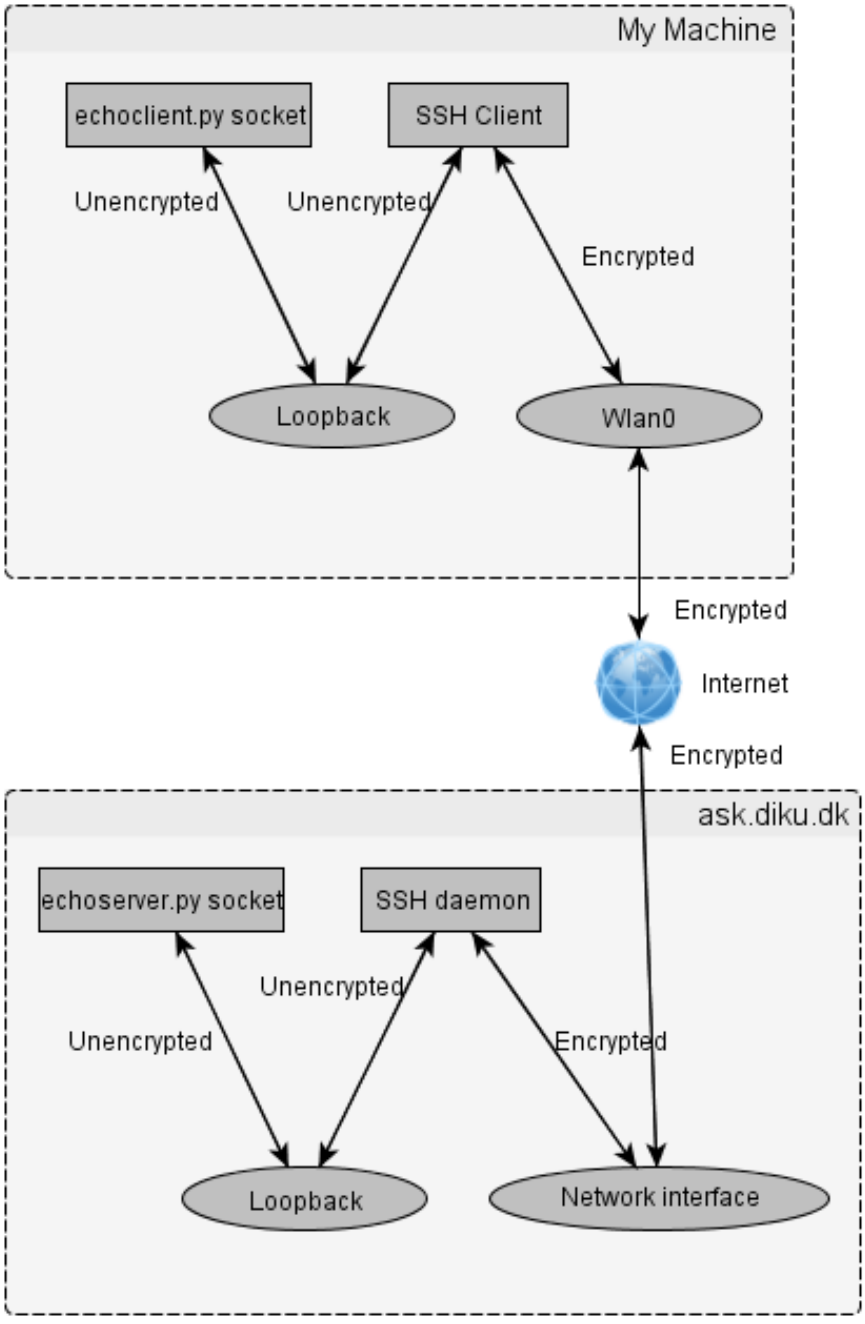
Q1:
The server is waiting for someone to connect on the "serversock". It is blocked at line 24 in the `socket.accept()` call.
Q2:
The port is different because the client socket connects TO port 6789, not from it. The FROM port is chosen by the system and not the program.
Q3:

I can see the messages because they're not encrypted or encoded in some other format. TCPdump just picks up the TCP packets coming on port 6789 and prints the content including TCP header. The content is plaintext, so it shows up in the terminal.

1.3.2 Part 2

Q4:
Í am unable to connect, there can be several reasons for this inability. The Í most likely is firewalling on DIKU prevent connections on ports that SCIENCE-Í IT do not use themselves, only allowing outbound connections on these ports. Also, `ping` is an *ICMP* packet and not a TCP packet like the socket connections in python. This means firewall treat the differently, ICMP is usually allowed. evne if all TCP ports are closed.
Q5:

The important bit is `6790:localhost:6002` it means that connection on our loopback interface that targets port 6790 should be directed through the SSH tunnel to port 6002 on the loopback interface at ask.diku.dk. So as long as both client and server sends to the loopback interface at the 2 specified ports, they can "talk" to each other.
Q6:
TCP dump output attached as q6.txt if needed
I can still see the output, the eason is the same as in Q3, we're sending stuff to the loopback interface in plaintext, therefore it is readable.
Q7:
Removing the `-i lo` switch removes the output because my machine defaults to the eth0 interface, which is not connected. Switching to *wlan0* also yields nothing since we're talking TO the loopback interface using port 6790.
Q7.5??:
Using my own public IP I can see a connection from my routers port 37456 to ask.diku.dk's port 22. Port 22 is commonly used for SSH services.
Q8:


My Machine
echoclient.py socket
SSH Client
Unencrypted
Unencrypted
Encrypted
Loopback
Wlan0
Encrypted
Internet
Encrypted
ask.diku.dk
echoserver.py socket
SSH daemon
Unencrypted
Unencrypted
Encrypted
Loopback
Network interface

2 Practical Section

2.1 Question 1

Since the program currently only listens to localhost (the loopback interface) any peer NOT on the machine will be unable to connect unless employing some sort of port forwarding like SSH tunneling.

Assuming a server that listens on an actual network interface and a network with NO firewall or other obstructions the following communication can take place:

```
Peer B <-> NS    : HANDSHAKE
Peer B --> NS    : LOOKUP Peer A
Peer B <-- NS    : INFO ...
Peer B <-> Peer A: HANDSHAKE
Peer B --> Peer A: MSG ...
Peer B <-- Peer A: MSG ACK
```

2.2 Question 2

I personally keep the socket alive, this means that no further handshakes are required for communicating with any of the present peers. The drawback is of course that I am blocking system resources that could maybe be used by other services. closing down sockets after a broadcast would free these resources, but since this is an academic excersice, and I know that the peers will not be connected in networks with 100.000s of peers I opted to keep them open.

2.3 Question 3

To allow multicasting, I would create a new dict with group ID's (key) and a list of member nicks (value). The group ID would then be negotiated between the peers when creating the group, and a new `GMSG` message implemented that send the group ID followed by the actual message.

Of course negotiating an ID that is not used can b tough in larger networks, and this model have drawbacks, if 6 peers create 2 seperate groups of 3 with same ID, the peers wil lbe unable to join the other groups because the ID for the dict is already in use.

2.4 Question 4

A *divide and conquer* method could be devised, we're the broadcasting peer once it recieves the userlist, picks a number of the peers and request that they help broadcasting the message. If the broadcasting peer sends the message to 2 peers, and the relay it to 2 peers each, the propegation time would be $O(log(n))$ instead. This means the the prorocal would need to be changed to allow this new command, and an algorithm for selecting peers that eliminate double transmissions (two peers write to the same peer) would need to be devised.

2.5 Question 5

We need the `MSG ACK` because there is no quarentee that the peer we're talking to as able to show the message, it could use a different format (version inconsistency?), or the other end might not be a peer at all, but another program using a similar but not identical protocol. The same goes for `BYE`, it means the peer will wait with closing the connection until the other peer have confirmed that it will remove the first peer from its dicts and lists.