Datanet 2013 Assignment 3

Danni Dromi Dídac Rodríguez Arbonès Hans Ienasescu Radu Dragusin
Marcos Vaz Salles

Deadline: 23:55 on May 21, 2013

This assignment is the third of four assignments to be handed in during this course. It is split into two parts, the first containing a set of theoretical exercises along with a activity exercise and the second detailing a practical assignment.

The assignment is due on May 21, 2013 at 23:55. It is to be handed in on the Absalon course page. It is your responsibility to make sure in time that the upload of your files succeeds. Email or paper submissions will not be accepted.

The assignment may be solved in groups of 2 or 3 students. We **strongly** encourage you to work in groups, and we have balanced the work load accordingly. Please contact your TA if you intend on solving it alone!

The assignment will be scored on a scale of 0-10 points. There will be **no resubmission** for this assignment. In order to participate in the exam, you must obtain a total of 24 points over the four assignments.
A well-formed solution to this assignment should include a PDF or plaintext file with answers to all exercises as well as questions posed in the programming part of the assignment, other formats will not be accepted. In addition, you must submit your code along with your written solution. The source code that is handed in should *only* contain `.py` files, no compiled python (`.pyc`) files. Your code should be compatible with Python version $\geq$ 2.6 && $<$ 3.0. If you have more than one source file please submit your assignment as a compressed folder, containing all source files.

Evaluation of the assignment will take both parts into consideration. Do not get hung up in a single question. It is best to make an effort on every question and write down all your solutions than to get a perfect solution for only one or two of the questions.

If you have questions about the exercises or the practical assignment, please use the discussion forum on Absalon so your fellow students may benefit from your questions.

1 Theoretical part

Each of the following sections deals with topics covered by the lectures and, to some extent, the TA sessions.

Each section contains a number of questions that should be answered **briefly** and **precisely**. Most of the questions can be answered in 2 sentences or less. We have annotated questions that demand longer answers, or figures with a proposed answer format. The activity exercise requires you to follow the described steps and, based on your outcomes of these steps, answer a number of questions.

Again, please note: Do write concise answers. You won't have time to do huge answers at the exam. Focus on identifying the core of each question.

Miscalculations are more likely to be accepted if you account for your calculations in your answers.

1.1 The Internet Protocol

1.1.1 Addresses and network masks

Part 1: This part deals with the meaning of network masks. Answer the following questions.

1. What is the purpose, from a router's point of view, of network masks? Or, asked another way, what is common for IP addresses in the same network? *(Answer with maximum 2 sentences)*

2. Explain how network masks may be expressed as an IP address. Is 255.225.255.0 a valid network mask? In the slash-notation, explain how /28 can be interpreted as a network mask? *(Answer with maximum 8 sentences)*

3. Explain the differences between *classful* and *classless interdomain routing* (CIDR) networks. *(Answer with 3 sentences or less)*

4. Explain how all information about a network can be derived from an IP address in the network and the network mask (in either the IP address style or slash-notation). *(Answer with 4 sentences or less. Hint: Information include broadcast addresses, the number of addresses available, etc)*

Part 2: In this part, you are to give the size of the network and give the network address, network mask, broadcast address, the first, 5th, and last usable IP address in the network (where applicable) in IP address style format.

1. Network: 130.225.165.16/27

2. Host: 10.0.42.23, netmask: 255.255.254.0

3. Host: 4.2.2.1/32

4. Host: 192.38.109.188/18

1.1.2 Network Address Translation

In the early days of the Internet, prior to the advent CIDR and wide scale consumer Internet access, a relative huge number of class A and B networks were given out to companies that had no need for that amount IP addresses. In order to make better use of the remaining addresses, a number of networks were reserved for private use, as specified by RFC1918[1].

These addresses can be used by *network-address translation*-enabled (NAT-enabled) routers. NAT works by translating the private IP address with a public IP address prior to forwarding packets onto the Internet.

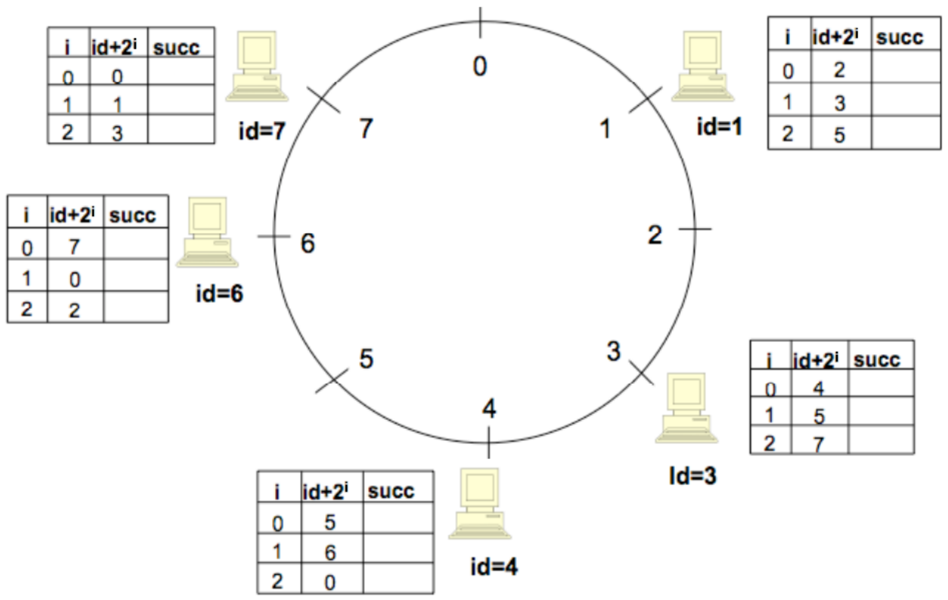
[1] `https://tools.ietf.org/html/rfc1918`

Part 1: Explain how NAT-enabled routers may handle multiple connections from multiple internal IP addresses. *(Answer with 4 sentences or less. Hint: How does the network stack distinguish between different processes running on the same IP address?)*

Part 2: Sketch a scenario, where multiple clients each with multiple long lived connections can exhaust the available network resources of a NAT-router regardless of the link capacity.

Part 3: Why is NAT a hack that breaks the layering principle of the Internet?

1.2 Distributed Hash Tables



Consider the above illustration of a Chord DHT [2], comprised of only 5 nodes. We show the finger tables for each node (i.e., the particular id distribution from which a node's routing table neighbors should be drawn), but we don't actually yet fill in the routing tables. Each node may be storing some items according to the Chord assignment rule (remember that Chord assigns keys to nodes in the same way as consistent hashing does, i.e., a successor mod N relationship).

1. Fill in the routing table for the node with id 1 above.
2. List the node(s) that will receive a query from node 1 for item 5 (i.e., the item named by key 5)
3. Suppose node 4 crashes, and the network converges to a new routing state. List the node(s) that will receive a query from node 7 for item 5.

[2] `http://www.cs.princeton.edu/courses/archive/spr11/cos461/docs/spring10-cos461-exam2.pdf`

1.3 SSH Tunneling

In this activity exercise you are given an implementation of an echo server and its corresponding client. The purpose of this activity is to set up the server and the client so that they can communicate with each other in different circumstances, and to observe the details behind this communication setup. The source code for the echo server and the client is provided in the files: `echoserver.py` and `echoclient.py`.Note that the server and the client are very simple and not very robust and the server can only handle one client at a time. In order to later differentiate the messages from the client to the server and from the server to the client we have modified the server response to contain the original client request along with the prefix `from_server:`. Thus, the protocol between the client and the echo server is:

1.3.1 ECHOSERVER ↔ CLIENT

`<MESSAGE>`

The client sends to the echoserver the message containing `<MESSAGE>` as data, and the server replies with:

`from_server:<MESSAGE>`

where `<MESSAGE>` contains the original data sent by the client. As an example we have:

Client sends to the echo server:
`hello!!!`
Server sends back to the client:
`from_server:hello!!!`

Part I In the first part we will set up the server and the client locally on your machine. You should use two terminal windows/tabs for each of the two components. First we will start the server (after we cd into the folder which contains the Python scripts).

```
$ python echoserver.py
waiting for connection...
```

At this point we can use netstat to check if the server is started. Given the default configuration of the server we have:

```
$ netstat -an | grep '6789\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:6789          0.0.0.0:*               LISTEN
```

Note: If you are on a Windows machine you unfortunately do not have grep capability. Thus you would have to try alternatives:

1. Log in on ASK and use your client and echo server there (you will need to do it later in the exercise anyway)
2. Use the command `netstat -an > file1.txt` which will write the output of `netstat -an` in the file `file1.txt`. After that you can use a text editor to search for the string `6789`

3. You can try and search for a “grep for Windows“ tool.

Question 1: What is the server doing at this point? Where in the code is the server blocked at this point?

We are going to start the client on the same machine and send some messages to the server.

```
$ python echoclient.py
>>hello
sent: hello
Received: from_server:hello
>>nice to see we are working :)
sent: nice to see we are working :)
Received: from_server:nice to see we are working :)
>>
```

Now that we have started the client and sent some messages successfully, we can check the status of the server and the client:

```
$ netstat -an | grep '6789\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:6789          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:6789          127.0.0.1:40542         ESTABLISHED
tcp        0      0 127.0.0.1:40542         127.0.0.1:6789          ESTABLISHED
```

Obs: You may have different values than 40542 on your machines, but if you have the default configuration of the server and the client, the rest should be the same.

Question 2: We also need to observe that the number 40542 (or in your case some other number) appears in both established sockets. Explain why.

We will use tcpdump to analyze the packets sent between the client and the server. This is achieved by typing the following command:

```
sudo tcpdump -Xs 1514 port 6789 -i lo
```

The option “-s“ is the snaplength (how much content are we grabbing) and the “-i lo“ specifies that we are listening to our loopback interface and the “port“ option specifies that we are only listening on port 6789 [3].

Note: Instead of tcpdump in Windows you can use Wireshark [4] or WinDump[5].

Question 3: Send some messages from the client and report the output of tcpdump. Can you see the actual messages in the output? Why?

[3]for more on tcpdump use man tcpdump or check out a tutorial at: `http://danielmiessler.com/study/tcpdump/`

[4]`http://www.wireshark.org/`

[5]`http://www.winpcap.org/windump/`

Recap on part I: Go through the same steps as above and report the outputs, but set up both the client and the server on ask.diku.dk (except tcpdump which does not work on ask because of administrative rights), [6] but change the configuration so that the server and the client communicate on port 6000.

Part II In the second part you will set up the configuration so that just the server runs on ask and the client runs locally. We know we can connect to ask using SSH but let's also check if we can ping ask.

```
$ ping ask.diku.dk
PING ask.diku.dk (130.225.96.225) 56(84) bytes of data.
64 bytes from ask.diku.dk (130.225.96.225): icmp_req=1 ttl=51 time=10.4 ms
64 bytes from ask.diku.dk (130.225.96.225): icmp_req=2 ttl=51 time=8.93 ms
64 bytes from ask.diku.dk (130.225.96.225): icmp_req=3 ttl=51 time=9.56 ms
```

So, we know and see ask from our own machines and therefore we know that we have a path to ask at the IP layer . Now start the server on ask on port 6001 and the client locally on your machine on the same port with a new host (ask.diku.dk or the ip of ask: 130.225.96.225).

Question 4: Can you establish the connection to the server? If yes, how? If not, explain why you cannot communicate with the server given that we know we have a path to ask at the IP layer, and we know there is a server listening on port 6001 on ask at the TCP layer.

At this point we will create an SSH tunnel from our local machine to ask.diku.dk and try to connect the client and the server with this new addition. Like before we want to keep the server on ask and the client on our local machine. We will set up the tunnel by running the following command on our local machine:

```
ssh -L 6790:localhost:6002 your_diku_user@ask.diku.dk
```

Question 5: Given the SSH tunnel that we set above what will be the client and server hosts and ports that you will define in your Python code in order to get a working communication? (hint: the host on the client is not ask.diku.dk or the ip of ask). Why can we communicate in this case?

If we look at the client side using netstat we have the following (remember that ports like 51082 will most likely be different on your machines) :

```
netstat -an | grep '6790\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:6790          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:6790          127.0.0.1:51082         ESTABLISHED
tcp        0      0 127.0.0.1:51082         127.0.0.1:6790          ESTABLISHED
tcp6       0      0 ::1:6790                :::*                    LISTEN
```

Let's tcpdump locally like above, but on the new port:

```
sudo tcpdump -Xs 1514 port 6790 -i lo
```

[6] use the command: ssh your_diku_user@ask.diku.dk

Question 6: Now send some messages from the client and report the output of tcpdump. Can you see the actual messages in the output? Why?

Given that we are now communicating with ask let's examine what connections we have with ask. We can do this by using netstat, again at the client side:

```
netstat -an | grep '130.225.96.225\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.0.0.7:52786          130.225.96.225:22       ESTABLISHED
```

Remember that the port 52786 will be different on your machines. Also, if you have more than one SSH connection to ask, you will see more than one entry in netstat, and in this case you should close all other SSH connections except for the tunnel.

Question 7: Run tcpdump on this port, but without the "-i lo" option, send some messages from the client and report the output. Can you see the actual messages in the output? Why?

In order to be able to use the same session on ask from which you started the server to write other commands, you can run the server in the background by pressing ctrl+z and then typing bg and enter. We are doing this because we only want a single connection to ask, so as to avoid confusion when we use netstat. Let's examine the ask side of the connection using netstat; on ask type:

```
netstat -an | grep '6002\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:6002          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:45873         127.0.0.1:6002          ESTABLISHED
tcp        0      0 127.0.0.1:6002          127.0.0.1:45873         ESTABLISHED
```

If you know your public ip you can check what are the connections from ask to your machine. Suppose our machines public ip is. 123.456.789.123, we have:

```
netstat -an | grep '123.456.789.123\|Address'
Proto Recv-Q Send-Q Local Address           Foreign Address           State
tcp        0      0 130.225.96.225:22       123.456.789.123:52786     ESTABLISHED
```

Try using netstat with your own public ip and see if you get similar outputs. Lookup port 22 on-line and see what service (relevant to this exercise) runs on this port.

Question 8 Given all the activity in part II draw a diagram of the client, server and SSH tunnel given that the client is on your local machine, the server is on ask and the tunnel is set up from your machine to ask as above. Draw all processes and sockets involved, as well as indicate where the connection is encrypted and where it is not.

2 Programming Part

For the practical part of assignment 3 you will be finishing the chat system by implementing the final features for peer-to-peer communication.

Currently, your system is able to connect several users to a centralized name server, and can from that point on allow them to look up other users and get a full userlist. This week you will make it possible for a user to send private messages to other users and to broadcast a message to all users connected to the service.

This programming assignment extends on your implementation from last week. The file `peer_exts.py` contains the functions your existing peer should handle. You should append these to your existing `peer.py` implementation.

This week you will be working mostly on implementing the peer, however you will still need to preform some minor adjustments to the name server.

For your convenience we will run a functional name server on one of the DIKU systems (further information will be made available on Absalon). This server is not constantly monitored, so, should it crash it may take some time for it to be restarted.

Since this week's implementation task focuses heavily on the peer you only have to hand that one in. If you are pressed for time you can use the name server we put up on `ask.diku.dk` instead of changing your own name server.

2.1 Peer-Peer Communication

This time you will mostly be extending the peer implementation. There are some minor changes you need to do to the name server, but more on that in Section 2.3.

Now that we wish to connect peers to one another, each peer will need a listening socket in order to listen for peers trying to connect to it (much like what the name server does).

When a user wishes to message another user, the peer will first have to establish a socket connection to the other peer. In order to do this the peer will need to query the name server for a look up on the information on the second peer. This information can now be used to initiate a socket connection to the peer.

When a peer wishes to connect to another peer, they must first preform an initial handshake (this is described in Appendix A.2). Afterwards (when connection has been established) they can send messages back and forth. When the peer receives a message it should be prompted to the user along with the nick of the sender (eg. `john:  how are you doing?`). Furthermore the receiving peer should send a response to the sending peer letting it know that the message was properly received (according to the protocol in Appendix A.2).
The chat session ends when one of the users decides to close their peer program by saying `/quit` (as shown in Appendix A.3). Upon leaving, the peer should send the appropriate messages to all connected peers (according to the protocol in Appendix A.2). Note that two connected peers can still communicate if the name server dies (of course they will need to have established the connection when the name server were still running).

2.2 Broadcasting

When broadcasting, a user wishes to sends a message to all other users on the system. First of all the peer will need to query the server for a list of active users and their listening sockets (you implemented the functionality for userlist last week). When the list is obtained, the peer will need to make sure that it has an open connection to all other connected peers. Then the peer can broadcast the message by sending it as a regular message to each of the peers. The receiving users should see this message like a regular message.

2.3 Name Server Extentions

Even though you implemented the full name server in the last assignment there are still some minor changes you will have to make to that implementation.

Now each peer has a listening socket on a specific port and the name server will need to store and distribute this information when a user wishes to message another user.

Therefore the peer - name server handshake must be extended to include also the peer's listening port (besides the user name). You will also need to change the methods that handle `LOOKUP` and `USERLIST` requests since these will now need to also return the listening port number (besides just the IP address).

Appendix A contains the full peer - name server protocol. The procedures for `HELLO`, `USERLIST` and `LOOKUP` have been changed from last assignment to accomodate the addition of the peer listening port.

Aside from implementing the client/server system you should answer the following questions:

Question 1

Given the following situation: say you are running a name server (using port 3456) and peer A (using port 1234) on the same machine M and you start another peer, B, on machine N. Can peer B on machine N connect with peer A on machine M? If the answer is yes, give a simple table that concisely shows the information that is exchanged between all three processes. If the answer is no, explain why the two peers cannot communicate and give a solution to this problem (i.e. what has to be changed?).

Example of information exchange:

```
peer B (bob) <--> name server: HANDSHAKE

peer B (bob) --> name server: LOOKUP marcos
peer B (bob) <-- name server: INFO ...

peer B (bob) <--> peer A (alice): HANDSHAKE

peer B (bob) --> peer A (alice): MSG
...
```

Question 2

Should the sockets used for broadcasting be kept alive by all peers or should each broadcast require new communication links to be established? What are the pros and cons of either approach?

Question 3

How should the protocol be changed if we want to add multicast (i.e. group chat)?
Note: For this question you are not expected to give a thorough description of how you would extend the existing protocol. You should explain in general terms how you could alter the current service to facilitate group chat. Use 1-2 paragraphs.

Question 4

For broadcasting to all users the protocol provides the command `USERLIST` at the server's side, which sends the names and information of all users to the requesting peer. This implies that the requesting peer has to do the actual broadcasting meaning the transmission time before everyone has received the message will be $O(n)$ (assuming each message can be sent at a constant rate and with n being the number of users).

How could the broadcast mechanism be changed so that the transmission time is kept in the magnitude of $O(\log(n))$?

Question 5

If you examine the protocols for `MSG` and `LEAVE` you will notice they both send a confirming response (either `200 MSG ACK` or `400 BYE`). However, since the service is using the TCP protocol which guarantees reliable delivery of packets, why do we need these confirming responses?

A Protocol

This appendix describes the `peer ↔ server` and `peer ↔ peer` protocols. The appendix will only document the new additions to the protocol, so features implemented in previous assignments will not figure here.

A.1 PEER ↔ NAME SERVER

`HELLO <nick> <port>`
To identify itself, a peer has to perform a simple 'handshake' protocol with the name server. The peer sends the command `HELLO` with the user's *nickname* and the *port number* of the peer's listening socket.
The name server may respond with:

`100 CONNECTED` Handshake successful, the peer is now connected to the name server.

`101 TAKEN` Handshake unsuccessful, the nickname was already taken. The name server directly closes the socket.

`102 HANDSHAKE EXPECTED` Received some information but not enough information for a proper handshake.

`LOOKUP <nick>`
Look up the address information for another peer identified by nickname.
The server may respond with:

`200 INFO <ip> <port>` The user was found and the ip-address is given along with the port number of the peer's listening socket.

`201 USER NOT FOUND` The user was not found.

`USERLIST`
Request a list of all users currently online according to the server.
The server may respond with:

`300 INFO <num peers> <nickname> <ip> <port>, ...` The first piece of information is the number of users that have registered. Following that is a comma separated list of *n* users. This list should not contain the peer doing the request.

`301 ONLY USER` You are the only peer currently registered.

`LEAVE`
Tell the server that the connection can be closed and the user is no longer available for chatting.
The server may respond with:

`400 BYE` Connection closed, user logged out.

`500 BAD FORMAT`
The name server should give this reply to any request sent that doesn't match the protocol.

A.2 PEER ↔ PEER

`HELLO <nick>`
Perform a handshake protocol with another peer.*nick* should be the nickname of the calling peer. This is quite similar to the handshake with the name server. The called peer may respond with:

`100 CONNECTED` Handshake was successful, and the calling peer is now connected to the called peer.

`101 REFUSED` The called peer has refused the connection.

`102 HANDSHAKE EXPECTED` Received some information but not enough information for handshake.

`MSG <message>`
Send a message to a peer. The argument should just be the message. The called peer should respond with:

`200 MSG ACK` A response indicating the message was received by the called peer. If the caller does not receive this response it should tell the user that the message wasn't delivered.

`LEAVE`
Tell the other peer that we wish to close the connection with the peer. The peer may respond with:

`400 BYE` Connection closed.

`500 BAD FORMAT`
The name server should give this reply to any request sent that doesn't match the protocol.

A.3 CLIENT ↔ USER

- /connect <IP> <PORT> - connect to name server and preform the handshake. IP should be the ip-address of the server and PORT the port that the server is listening on.The handshake is described in A.1
  The user should be prompted a message telling whether or not the connection succeded.
- /nick <NAME> - select a nick name. The user should get a confirming message. This command does not invoke any communication with the server. The nickname should not contain commas (',‘), because they will interfere with the protocol for USERLIST.

- /msg <NICK> <MSG> send a private message to the user with nick name NICK. The receiver should see the message like this:

  ```
  Eric: Have you started on the Datanet assignment yet?
  ```

  When this command is called the peer should check if there is a connection open to the peer of NICK. If there is it can use it to send the message and if not it will need to first query the name server for information before handshaking the other peer. The sender should only get some output if the message wasn't received.

- /all <MSG> - broadcast a message. The receivers should see this message as if sent like a regular message. The peer will need to query the name server for the userlist before being able to send the message to all connected users. Once again the user should only get some output if the broadcasting failed.

- /userlist - get a list of online users A.1. The user should get a list looking like:

  ```
  Online Users:

  Eric - You
  Pete - 192.32.5.1
  Shawn - 64.128.10.56
  Lisa - 56.0.23.2.1
  ```

- /leave - close the connection with the name server. The user should get a confirming message A.1.

- /quit - close the peer application. The user should get a confirming message. This command does not invoke any communication with the server.