UPush Chat Service

Introduction

Chat services have existed since the first days of the Internet, just like SMS has existed since the first days of digital mobile telephony. Chat provides a simple service that allows people to communicate at their own leisure, permitting but not demanding immediate responses. It is unobtrusive compared to phone or video calls, and much more light-weight compared to email.

Such a simple service is not necessarily reliable, but users expect to know whether a message has been received at its destination or not. Messages should not be stored on disk in case the receiver is offline.

The Task

In this assignment, you are going to write a UDP-based chat service called UPush. The chat service is very fast and simple, and neither reliable nor secure. It is inspired by real-world chat services where users register their chat processes at a well-known server in order to be found, but where messages between clients are sent directly without passing through a server. All commands and all messages are human-readable, clear-text messages encoded in ASCII. The server is inspired by DNS servers, but instead of mapping hostnames to IP addresses it maps nicknames to IP addresses and ports.

Packet-loss is to be emulated using the given precode, and packet loss has to be handled by both the clients and the server. In all of your code, you should not send UDP packets using library functions, but you should use the precode function `send_packet()`.

Registration and Lookup

The **UPush server** is the first element of the UPush chat service to start. The server does not store information about clients on disk, only in memory. It waits forever for messages from UPush clients. These messages can either be registration messages or lookup messages. The UPush server takes a port number and a loss probability as command line arguments.

```
> ./upush_server <port> <loss_probability>
```

`port` is the port number that the server uses to receive and send UDP packets (datagrams). If the given port cannot be used (perhaps it is already in use by another process), the server must print an error message and quit.

`loss_probability` is a value between 0 and 100 that gives the percentage of packets that should be discarded by the precode function `send_packet()`. The percentage must be converted by you to a probability value between 0 and 1, and you must call the precode function `set_loss_probability()` with this probability value in main() before sending any packets.

Registration

When a client sends a registration message, it means that the client wants to be found by other clients using a nickname ("nick"). The client sends just the registration command with its nick to the server, by sending a packet containing a string of the form "`PKT number REG nick`". This nickname can only consist of ASCII characters, it cannot be longer than 20 bytes, and it cannot contain white space (space, tab, return).

The number is the sequence number for a stop-and-wait protocol that you implement for communication between the client and the server (however, unlike normal stop-and-wait, you are allowed to use other sequence numbers than just 0 and 1). When the server responds to a client's registration message with sequence number M, the server's response message uses the sequence number M as well.

To register a client, the server needs to know the client's IP address and port number. The server determines the client process' IP address and port from information provided by the `recvfrom()` function that it uses to receive the message from its UDP socket. The server keeps such registrations in memory in a suitable data structure (perhaps a list or an array). When a nick is already registered, the old IP/port information is simply replaced by the new IP/port. The UPush server confirms the registration by sending an OK message of the form "`ACK number OK`" to the client.

Lookup

When a client sends a lookup message, it contains a nick. The format of the lookup message is "`PKT number LOOKUP nick`". The server checks if this nick is registered. If it is, the server returns the nick, its IP address and port to the client that sent the lookup message. A successful lookup should result in a message on the form "`ACK number NICK nick IP address PORT port`". If it is not registered, the server returns a not-found message, which should be on the form "`ACK number NOT FOUND`".

A **UPush client** is a program that takes the user's nick, the IP address and port of a UPush server, a timeout value in seconds and a loss probability on the command line. When the UPush client starts, it tries to register its nick with the server immediately (i.e. by sending "`PKT number REG nick`"). It waits for either an OK message from the server or a timeout. If the timeout comes first, the UPush client prints an error message and exits. If the client receives an OK message, it enters an event loop to wait for both UDP packets and user input, at the same time. This event loop is called the **main event loop**.

```
> ./upush_client <nick> <ip-address> <port> <timeout> <loss_probability>
```

`nick` is the client's nick.

`ip-address` and `port` are the server's IP address and port.

`timeout` is the time in seconds between a client's action (like sending a packet) and when the client assumes that the consequence of this action (like receiving an ACK packet) will not happen. It is used whenever the client should wait until it retransmits something or until it gives up on something.

`loss_probability` is the same as for the server. The client must call the precode function `set_loss_probability()` with the probability computed from the given percentage before sending a packet for the first time.

Messaging

Input from a socket

When a UDP packet from another client is received, the client expects that the payload of the packet has the form “`PKT number FROM from_nick TO to_nick MSG text`”. The client must verify two conditions:

- It must verify that the format is correct (more details below).
- If that is the case, it must verify that `tonick` is identical to its own nick.

If both conditions are fulfilled, it prints "`from_nick: text`" to standard output (`stdout`). It must also send a message back to the sender confirming that the message was received with the following format "`ACK number OK`". If one of the two conditions is not fulfilled, it prints an error message to standard error (`stderr`). If this happens the client should respond with "`ACK number WRONG NAME`" or "`ACK number WRONG FORMAT`" depending on the failing condition. If both conditions fail you are free to choose one of them.

The client needs to be able to detect duplicates. The client should use the stop-and-wait protocol for this (except that you are not limited to sequence numbers 0 and 1). When stop-and-wait is used, there is at most one outstanding message for a client at any given time. This means that the client only needs to store the last number received from every other client. Duplicate messages should be answered like the original, but they should not be printed.

Input from `stdin`

When user input is received from standard input (`stdin`), the client expects that it consists of the word "`QUIT`" or a line containing a message, with the format "`@tonick text`". An input ends when the user presses return (the newline should **not** be part of the message). The maximum length of `text` should be 1400 bytes. If the user input is longer, the client should ignore further bytes.

If the input is "`QUIT`", the client releases all resources (closing sockets, freeing memory, etc.) and terminates.

If the input has the form "`@tonick text`", the client checks first if it already knows the IP address and port for the nick `tonick`. It does this by looking it up in a **nick-cache** in the client's memory (this cache is empty when the client starts). If that is not the case, the client enters a lookup procedure. It is allowed that the client blocks (ignore arriving packets and ignore input text from the user) until this lookup procedure is finished. To perform a lookup, the client sends a message of the form "`PKT number LOOKUP tonick`" to the UPush server. Three things can happen:

1. The client does not receive a response from the server within the timeout period. It repeats the lookup attempt 2 more times, then it prints an error message to `stderr` and performs the QUIT procedure.
2. The client receives a message of the form "`ACK number NOT FOUND`", where the number is a copy of the number that the client sent. The client prints the error message "`NICK tonick NOT REGISTERED`" to `stderr`. It returns to the main event loop.

3. The client receives a message of the form "`ACK number NICK tonick IP address PORT port`". The client stores this information in memory in its **nick-cache.**

Once the sender knows the receiver's IP address and port number, either from its own cache or via the server, it sends its message directly to the address/port pair.

When a client sends a message, it must handle the possible loss of either the message itself or the corresponding acknowledgement. It uses a protocol inspired by stop-and-wait protocol to do this (you are allowed to use other numbers than just 0 and 1), but with an additional lookup step:

- When a message is sent and no ACK is received before `timeout` seconds (the value of `timeout` comes from the parameter of the `upush_client` command), the client repeats the transmission once.
- If the second transmission fails, the client performs the lookup operation for the nick again (because the other client may have registered a new IP/port in the meantime).
- If the lookup operation fails, the client prints the message "`NICK tonick NOT REGISTERED`" to `stderr`. It stops its attempts to deliver the message.
- If the lookup operation is successful, the client attempts to send the message two more times, but now to the newly received IP/port pair.
- If this fails as well, the client prints the message "`NICK tonick UNREACHABLE`" to `stderr`.

Sending several messages and messages to several nicks

Whether you choose to store the messages in a linked list or read from `stdin` on a per need basis is up to you. Other data structures for storing messages are also fine. It might be difficult to read on a per-need basis when the client communicates with multiple clients, so we recommend that messages are read and stored in the client'. Note that this has to be done separately for each receiver. In other words, each receiver that the client communicates with needs to have its own expected number and queue of messages.

Heartbeat

All clients quit without informing the server. To prevent stale entries from accumulating over time, clients have to send a registration message once every 10 seconds.

The server should not use any registration that has not been updated in the last 30 seconds to respond to a client's lookup request. The server should return "`ACK number NOT FOUND`" instead and remove the nick from the server's list of registered nicks at a suitable time.

Blocking and Unblocking Users

Clients should be able to block nicks from whom they do not want to receive messages. Packets from a blocked nick are received and their packets are parsed, but nothing is printed on `stderr` or `stdout`. Blocking a nick will prevent the client from sending messages to that nickname, too. Whenever a message is sent or received, the client must ensure that the nick is not on the block list before processing the message.

To add a nick to the block list, the user must type "`BLOCK nick`" on `stdin`. To remove a nick from the block list, the user must issue "`UNBLOCK nick`" on `stdin`. Blocking and unblocking are entirely local operations that are handled inside the client.

Documentation

Write a design document that explains your design choices as well as the code. Explain what works, and if parts of the task have not been solved successfully, clarify in which way they fail. Your documentation must be a PDF file and it should not be longer than 1 page (10pt font or bigger).

Message summary

All messages are ASCII messages, including the text in the text message. That means that you will not be able to use characters such as ø og å. Linux offers several helper functions to check if a `char` contains an ASCII character, a number or a whitespace (space, tab, return): `isalpha(c)`, `isnum(c)`, `isspace(c)` and more.

The following table gives an overview of the message types and their formats. The words that are all capital letters should not be changed. The words that are all lowercase can and should be changed. The presence of `[ ]` indicates that the word is optional. For example, a positive text response will be "`ACK number OK`", but if there is an error with the format should be "`ACK number WRONG FORMAT`".

Message type	**`Format`**
Registration message	"`PKT number REG nick`"
Registration ok	"`ACK number OK`"
Lookup message	"`PKT number LOOKUP nick`"
Lookup fail response	"`ACK number NOT FOUND`"
Lookup success response	"`ACK number NICK nick IP address PORT port`"
Text message	"`PKT number FROM fromnick TO tonick MSG text`"
Text response	"`ACK number [OK/WRONG NAME/WRONG FORMAT]`"

Advice

To check the program for memory leaks, we recommend that you run Valgrind with the following flags:

```
valgrind \
    --track-origins=yes \
    --malloc-fill=0x40 \
    --free-fill=0x23 \
    --leak-check=full \
    --show-leak-kinds=all \
    DITT_PROGRAM
```

Netcat

Netcat is a program that can be used to send and receive UDP messages to and from a specific port. You can run netcat in the following way:

```
> nc -u <ip-address> <port>
```

This can help you debug and test your server and clients before everything is fully implemented. What you enter after the invocation of the command will be sent as a UDP-packet to the receiver. If the program terminates with an error it is likely to be caused by nobody listening to the given port on the given ip-address. Make sure the server is listening to the correct port on the correct machine.

Development steps

One possible way to solve this task is in the following steps. We recommend that you set the loss probability to 0 until you start to test that the loss recovery and re-transmissions work.

1. Create the server
    a. Read parameters from the command line and set the packet loss probability by calling `set_loss_probability()`.
    b. Create data structures for registered clients
    c. Create a socket and try to send registration and lookup messages using netcat (just to see that the receiving works)
    d. Implement registration. Add information to the data structure in response to registration messages
    e. Implement lookup. Respond to lookups using the data structure and the information in the request
    f. Test it using print-statements and netcat to send registration and lookup messages
2. Create the client
    a. Create a socket and try to send something to the server
    b. Implement registration.
    c. Implement a lookup of a known (hard-coded) nick and store the information in an appropriate data structure. This is a step that is intended to make it easier to detect errors early
    d. Implement sending and response to messages. Try to send data to a known nick using the information stored. Implement a response to incoming messages
    e. Implement parsing of stdin so that you can replace the known nick with user-given nick. This will involve doing lookups to get the necessary information
    f. Make the client able to listen to the socket and stdin at the same time
3. Implement timeout for loss recovery for both the server and the client. This is the point that loss probability should be set to a value higher than 0.

    a. Implement stop-and-wait for registration messages. Use `select` to have a timeout for retries.
    b. Implement stop-and-wait for text messages. Extend the main event loop to have a timeout for retries.
4. Implement heartbeat messages
    a. Implement on the server and test with netcat
    b. Implement on the client
5. Implement blocking of sending and receiving from specified nicks

Submission

You must submit all your code in a single TAR, TGZ or ZIP archive.

Include your Makefile and include the precode.

If your file is called `<candidatenumber>.tar` or `<candidatenumber>.tgz`, we will use the command tar on `login.ifi.uio.no` to extract it. If your file is called `<candidatenumber>.zip`, we will use the command unzip on login.ifi.uio.no to extract it. Make sure that this works before uploading the file. It is also prudent to download and test the code after delivering it.

Your archive must contain a Makefile, which will have at least these options:

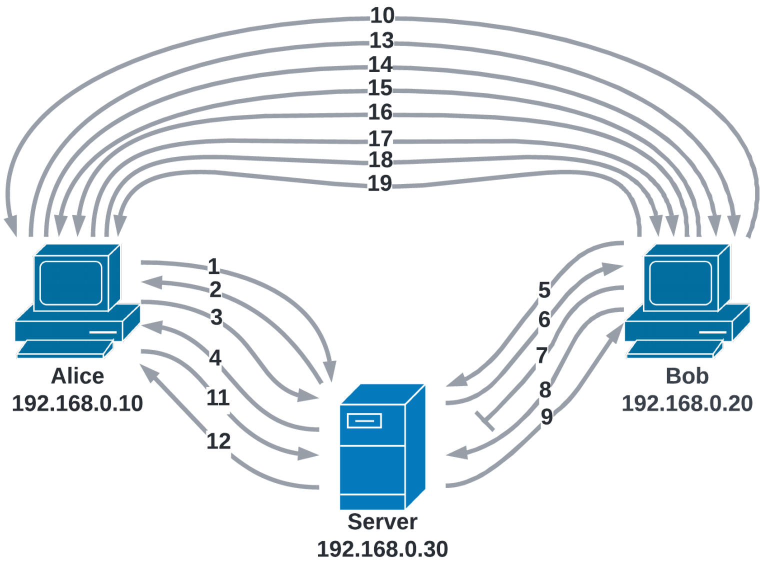
- `make` - compiles both your programs into executable binaries `upush_server` and `upush_client`
- `make all` - does the same as make without any parameter
- `make clean` - deletes the executables and any temporary files (e.g. `*.o`)

About the Evaluation

The home exam will be evaluated on the computers of the login.ifi.uio.no pool. The programs must compile and run on these computers. If there are ambiguities in the assignment text you should point them out and write your choices and assumptions in a separate document and deliver it alongside the code.

Appendix: Example Scenario

This appendix shows how a particular scenario plays out, involving message exchange between two clients and a server on a local area network. First, Alice registers. She tries to look up Bob, but he is not yet online. Then, Bob registers and tries to look up Alice. However, his second packet mysteriously disappears *en route*. Luckily, the first automatic retransmission gets through. Having obtained Alice´s address, Bob sends her a message. She can reply after a second lookup – this time with success. Finally, they exchange a couple of messages. Note that there are three pairs of communicating parties, but four series of sequence numbers since, in contrast to TCP, UPush does not establish a connection between the parties.



```
1.  [Alice->Server]    PKT 0 REG Alice
2.  [Server->Alice]    ACK 0 OK
3.  [Alice->Server]    PKT 1 LOOKUP Bob
4.  [Server->Alice]    ACK 1 NOT FOUND
5.  [Bob->Server]      PKT 0 REG Bob
6.  [Server->Bob]      ACK 0 OK
7.  [Bob->Server]      PKT 1 LOOKUP Alice
8.  [Bob->Server]      PKT 1 LOOKUP Alice
9.  [Server->Bob]      ACK 1 NICK Alice IP 192.168.0.10 PORT 42394
10. [Bob->Alice]       PKT 0 FROM Bob TO Alice MSG Good morning, Alice!
11. [Alice->Server]    PKT 0 LOOKUP Bob
12. [Server->Alice]    ACK 0 NICK Bob IP 192.168.0.20 PORT 18690
13. [Alice->Bob]       ACK 0 OK
14. [Alice->Bob]       PKT 0 FROM Alice TO Bob MSG And to you, Bob!
15. [Bob->Alice]       ACK 0 OK
16. [Bob->Alice]       PKT 1 FROM Bob TO Alice MSG Coffee or tea?
17. [Alice->Bob]       ACK 1 OK
18. [Alice->Bob]       PKT 1 FROM Alice TO Bob MSG Coffee, please!
19. [Bob->Alice]       ACK 1 OK
```