

# English

## 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's. 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	<code>"PKT number REG nick"</code>
Registration ok	<code>"ACK number OK"</code>
Lookup message	<code>"PKT number LOOKUP nick"</code>
Lookup fail response	<code>"ACK number NOT FOUND"</code>
Lookup success response	<code>"ACK number NICK nick IP address PORT port"</code>
Text message	<code>"PKT number FROM fromnick TO tonick MSG text"</code>
Text response	<code>"ACK number [OK/WRONG NAME/WRONG FORMAT]"</code>

## 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 `<candidate number>.tar` or `<candidate number>.tgz`, we will use the command `tar` on `login.ifi.uio.no` to extract it. If your file is called `<candidate number>.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:

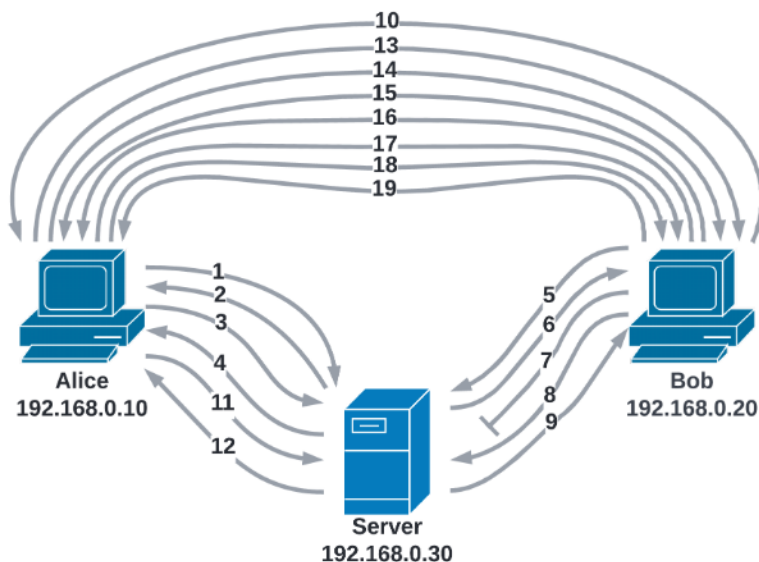
- `make` - compiles both your programs into executable binaries `upush_server` and `upush_client`
- `make all` - does the same as `make` without any parameter
- `make clean` - deletes the executables and any temporary files (e.g. `*.o`)

## 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



# Norsk

## UPush chattjeneste

### Introduksjon

Chattjenester har eksistert siden internettets første dager, akkurat som SMS har eksistert siden den digitale mobiltelefoniens første dager. Chat gir en enkel tjeneste som lar folk kommunisere på egenhånd, og tillater, men krever ikke, umiddelbare svar. Den er diskret sammenlignet med telefon- eller videosamtaler, og langt mer lettvektig sammenliknet med e-post.

En slik enkel tjeneste er ikke nødvendigvis pålitelig, men brukere forventer å vite om en melding er mottatt på destinasjonen eller ikke. Meldinger skal ikke lagres på disk dersom mottakeren er frakoblet.

### Oppgaven

I denne oppgaven skal du skrive en UDP-basert chattjeneste kalt UPush. Chattjenesten er veldig rask og enkel, og verken pålitelig eller sikker. Den er inspirert av chattjenester i den virkelige verden der brukere registrerer chatteprosessene sine på en kjent server for å bli funnet, men hvor meldinger mellom klienter sendes direkte uten å gå gjennom serveren. Alle kommandoer og alle meldinger er menneskelesbare klartekstmeldinger kodet i ASCII. Serveren er inspirert av DNS-servere, men i stedet for å koble vertsnavn til IP adresser, tilordner den kallenavn til IP-adresser og porter.

Pakketap skal emuleres ved å bruke den gitte prekoden, og pakketap må håndteres av både klientene og serveren. For sending av UDP pakker skal du ikke bruke biblioteksfunksjoner, men prekodefunksjonen `send_packet()`.

### Registrering og oppslag

**UPush-serveren** er det første elementet i UPush-chattjenesten som starter. Serveren lagrer ikke informasjon om klienter på disken, kun i minnet. Den venter evig på meldinger fra UPush-klienter. Disse meldingene kan enten være registreringsmeldinger eller oppslagsmeldinger. UPush-serveren tar et portnummer og en tapssannsynlighet som kommandolinjeargumenter.

```
> ./upush_server <port> <tapssannsynlighet>
```

`port` er portnummeret som serveren bruker for å motta og sende UDP-pakker (datagrammer). Hvis den gitte porten ikke kan brukes (kanskje den allerede er i bruk av en annen prosess), må serveren skrive ut en feilmelding og avslutte.

`tapssannsynlighet` er en verdi mellom 0 og 100 som gir prosentandelen av pakker som skal forkastes av prekodefunksjonen `send_packet()`. Prosentandelen må konverteres av deg til en

sannsynlighetsverdi mellom 0 og 1, og du må kalle prekodefunksjonen `set_loss_probability()` med denne sannsynlighetsverdien i `main()` før du sender noen pakker.

## Registrering

Når en klient sender en registreringsmelding, betyr det at klienten ønsker å bli funnet av andre klienter som bruker et kallenavn ("nick"). Klienten sender bare registreringskommandoen med sitt nick til serveren, ved å sende en pakke som inneholder en streng på formen "PKT nummer REG nick". Dette kallenavnet kan bare bestå av ASCII-tegn, det kan ikke være lengre enn 20 byte, og det kan ikke inneholde whitespace (mellomrom, tabulator, retur).

Nummeret er sekvensnummeret for en stopp-og-vent-protokoll som du implementerer for kommunikasjon mellom klienten og serveren (i motsetning til vanlig stopp-og-vent har du imidlertid lov til å bruke andre sekvensnumre enn bare 0 og 1). Når serveren svarer på en klients registreringsmelding med sekvensnummer M, bruker serverens svarmelding også sekvensnummeret M.

For å registrere en klient må serveren kjenne klientens IP-adresse og portnummer. Serveren ekstraherer klientprosessens IP-adresse og port fra informasjon gitt av `recvfrom()`-funksjonen, som den bruker for å motta meldingen fra UDP-socketen. Serveren oppbevarer slike registreringer i minnet i en passende datastruktur (kanskje en liste eller en array). Når en nick allerede er registrert, erstattes den gamle IP-/portinformasjonen med den nye IP-adressen/-porten. UPush-serveren bekrefter registreringen ved å sende en OK-melding på formen "ACK nummer OK" til klienten.

## Oppslag

Når en klient sender en oppslagsmelding, inneholder den en nick. Formatet på oppslagsmeldingen er "PKT nummer LOOKUP nick". Serveren sjekker om denne nicken er registrert. Hvis det er det, returnerer serveren nicken, IP-adressen og porten til klienten som sendte oppslagsmeldingen. Et vellykket oppslag skal resultere i en melding på formen "ACK nummer NICK nick IP adresse PORT port". Hvis den ikke er registrert, returnerer serveren en «ikke funnet»-melding, som har formen "ACK nummer NOT FOUND".

En **UPush-klient** er et program som tar brukerens nick, IP-adressen og porten til en UPush-server, en *timeout*-verdi i sekunder og en tapssannsynlighet på kommandolinjen. Når UPush-klienten starter, prøver den å registrere nicken sin på serveren umiddelbart (dvs. ved å sende "PKT nummer REG nick"). Den venter enten på en OK-melding fra serveren eller på tidsavbrudd (timeout). Hvis tidsavbruddet kommer først, skriver UPush-klienten ut en feilmelding og avslutter. Hvis klienten mottar en OK-melding, går den inn i en hendelsesløkke for å vente samtidig på både UDP-pakker og brukerinndata. Denne hendelsesløkken kalles **hovedhendelsesløkken**.

```
> ./upush_client <nick> <adresse> <port> <timeout> <tapssannsynlighet>
```

`nick` er klientens kallenavn.

`adresse` og `port` er serverens IP-adresse og port.

`timeout` er tiden i sekunder mellom en klients handling (som å sende en pakke) og når klienten antar at konsekvensen av denne handlingen (som å motta en ACK-pakke) ikke vil skje. Den brukes når klienten skal vente til den sender noe på nytt eller til den gir opp noe.

`tapssannsynlighet` har samme betydning her som for serveren. Klienten må kalle `prekode-funksjonen` `set_loss_probability()` med sannsynligheten beregnet fra den gitte prosenten før en pakke sendes for første gang.

## Meldingsutveksling

### Input fra `socket`

Når en UDP-pakke fra en annen klient mottas, forventer klienten at nyttelasten til pakken har formen "PKT nummer FROM `fra_nick` TO `til_nick` MSG tekst". Klienten må verifisere to forhold:

- Den må verifisere at formatet er korrekt (mer detaljer nedenfor).
- Hvis det er tilfelle, må den verifisere at `til_nick` er identisk med sin egen `nick`.

Hvis begge betingelsene er oppfylt, skriver den ut "`fra_nick: tekst`" til *standard output* (`stdout`). Den må også sende en melding tilbake til avsenderen som bekrefter at meldingen ble mottatt med følgende format "ACK nummer OK". Hvis én av de to betingelsene ikke er oppfylt, skriver den ut en feilmelding til *standard error* (`stderr`). Hvis dette skjer, bør klienten svare med "ACK nummer WRONG NAME" eller "ACK nummer WRONG FORMAT" avhengig av feiltilstanden. Hvis begge vilkårene mislykkes, står du fritt til å velge én av dem.

Klienten må kunne oppdage duplikater. Klienten bør bruke stopp-og-vent-protokollen for dette (bortsett fra at du ikke er begrenset til sekvensnumrene 0 og 1). Når stopp-og-vent brukes, er det maksimalt én utestående melding for en klient til enhver tid. Dette betyr at klienten kun trenger å lagre det siste nummeret mottatt fra hver av de andre klientene. Dupliserte meldinger skal besvares som originalen, men ingenting skal skrives ut.

### Input fra `stdin`

Når brukerinput mottas fra standard input (`stdin`), forventer klienten at den består av ordet "QUIT" eller en linje som inneholder en melding, på formatet "@`til_nick` tekst". En inntasting avsluttes når brukeren trykker return (nylinjen skal **ikke** være en del av meldingen). Maksimal lengde på tekst bør være 1400 byte. Hvis brukerinndata er lengre, bør klienten ignorere ytterligere bytes.

Hvis input er "QUIT", frigjør klienten alle ressurser (lukker sockets, frigjør minne, etc.) og avslutter.

Hvis input har formen "@`til_nick` tekst", sjekker klienten først om den allerede kjenner IP-adressen og porten for kallenavnet `til_nick`. Den gjør dette ved å slå den opp i en **nick-cache** i klientens minne (denne cachen er tom når klienten starter). Hvis det ikke er tilfelle, går klienten inn i en oppslagsprosedyre. Det er tillatt at klienten blokkerer (ignorer innkommende pakker og ignorer inputtekst fra brukeren) til denne oppslagsprosedyren er fullført. For å utføre et oppslag, sender klienten en melding på formen "PKT nummer LOOKUP `til_nick`" til UPush-serveren. Tre ting kan skje:

1. Klienten mottar ikke svar fra serveren innen tidsavbruddsperioden. Den gjentar oppslagsprosedyren 2 ganger til før den skriver ut en feilmelding til `stderr` og utfører QUIT-prosedyren.

2. Klienten mottar en melding på formen `"ACK nummer NOT FOUND"`, hvor nummeret er en kopi av nummeret som klienten sendte. Klienten skriver ut feilmeldingen `"NICK til_nick NOT REGISTERED"` til `stderr`. Den går tilbake til hovedhendelsesløkken
3. Klienten mottar en melding på formen `"ACK nummer NICK til_nick adresse PORT port"`. Klienten lagrer denne informasjonen i minnet i sin **nick-cache**.

Når avsenderen kjenner mottakerens IP-adresse og portnummer, enten fra sin egen cache eller via serveren, sender den sin melding direkte til adresse/port-paret.

Når en klient sender en melding, må den håndtere det mulige tapet av enten selve meldingen eller den tilsvarende bekreftelsen. Den bruker en protokoll inspirert av stopp-og-vent-protokollen for å gjøre dette (du har lov å bruke andre tall enn bare 0 og 1), men med et ekstra oppslagstrinn:

- Når en melding sendes og ingen ACK mottas før `timeout` sekunder (verdien for `timeout` er tatt fra parameteren til `upush_client`-kommandoen), gjentar klienten overføringen én gang.
- Hvis den andre overføringen mislykkes, utfører klienten oppslagsoperasjonen for nicken igjen (fordi den andre klienten kan ha registrert en ny IP/port i mellomtiden).
- Hvis oppslagsoperasjonen mislykkes, skriver klienten ut meldingen `"NICK til_nick NOT REGISTERED"` til `stderr`. Den stopper sine forsøk på å levere meldingen.
- Hvis oppslagsoperasjonen er vellykket, prøver klienten to ganger til å sende meldingen, men nå til det nye mottatte IP/port-paret.
- Hvis dette også mislykkes, skriver klienten ut meldingen `"NICK til_nick UNREACHABLE"` til `stderr`.

## Sende flere meldinger og meldinger til flere nick

Om du velger å lagre meldingene i en lenket liste eller lese fra `stdin` etter behov er opp til deg. Andre datastrukturer for lagring av meldinger er også fine. Det kan være vanskelig å lese etter behov når klienten kommuniserer med flere klienter, så vi anbefaler at meldinger leses og lagres i klienten. Merk at dette må gjøres separat for hver mottaker. Med andre ord må hver mottaker som klienten kommuniserer med ha sitt eget forventede antall og kø av meldinger.

## Hjerteslag

Alle klienter avslutter uten å informere serveren. For å forhindre at foreldede oppføringer akkumuleres over tid, må klienter sende en registreringsmelding hvert 10. sekund.

Serveren skal ikke bruke noen registrering som ikke har blitt oppdatert i løpet av de siste 30 sekundene til å svare på en klients oppslagsforespørsel. Serveren bør returnere `"ACK nummer NOT FOUND"` i stedet og fjerne nicket fra serverens liste over registrerte nicks på et passende tidspunkt.

## Blokkering og avblokkering av brukere

Klienter skal kunne blokkere nicker som de ikke ønsker å motta meldinger fra. Pakker fra en blokkert nick mottas og pakkene deres analyseres, men ingenting skrives ut på `stderr` eller `stdout`. Blokkering av en nick forhindrer også at klienten selv sender meldinger til den blokkerte nicken. Når en melding sendes eller mottas, må klienten sørge for at nicken ikke er på blokkeringslisten før meldingen behandles.

For å legge en nick til blokkeringslisten, må brukeren skrive `"BLOCK nick"` på `stdin`. For å fjerne en nick fra blokkeringslisten, må brukeren skrive `"UNBLOCK nick"` på `stdin`. Blokkering og oppheving av blokkering er helt lokale operasjoner som håndteres inne i klienten.

## Dokumentasjon

Skriv et designdokument som forklarer dine designvalg samt koden. Forklar hva som fungerer, og hvis deler av oppgaven ikke er løst vellykket, forklar på hvilken måte de mislykkes. Dokumentasjonen din må være en PDF-fil, og den bør ikke være lengre enn 1 side (10 pkt skrift eller større).

## Meldingssammendrag

Alle meldinger er ASCII-meldinger, inkludert teksten i tekstmeldingen. Det betyr at du ikke vil kunne bruke tegn som `ø` og `å`. Linux tilbyr flere hjelpefunksjoner for å sjekke om en `char` inneholder et ASCII-tegn, et tall eller et mellomrom (mellomrom, tabulator, retur): `isalpha(c)`, `isnum(c)`, `isspace(c)` og mer.

Tabellen nedenfor gir en oversikt over meldingstypene og deres formater. Ordene som alle er store bokstaver bør ikke endres. Ordene som alle er små kan og bør endres. Tilstedeværelsen av `[ ]` indikerer at ordet er valgfritt. For eksempel vil en positivt tekstrespons være `"ACK nummer OK"`, men hvis det er en feil med formatet skal det være `"ACK nummer WRONG FORMAT"`.

Meldingstype	Format
Registreringsmelding	<code>"PKT nummer REG nick"</code>
Registrering OK	<code>"ACK nummer OK"</code>
Oppslagsmelding	<code>"PKT nummer LOOKUP nick"</code>
Oppslagssvar ved feil	<code>"ACK nummer NOT FOUND"</code>
Oppslagssvar ved suksess	<code>"ACK nummer NICK nick adresse PORT port"</code>
Tekstmelding	<code>"PKT nummer FROM fra_nick TO to_nick MSG tekst"</code>
Tekstrespons	<code>"ACK nummer [OK/WRONG NAME/WRONG FORMAT]"</code>

## Råd

For å sjekke programmet for minnelekkasjer anbefaler vi at du kjører Valgrind med følgende flagg:

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

## Netcat

Netcat er et program som kan brukes til å sende og motta UDP-meldinger til og fra en bestemt port. Du kan kjøre Netcat på følgende måte:

```
> nc -u <adresse> <port>
```

Dette kan hjelpe deg med å feilsøke og teste serveren og klientene før alt er fullt implementert. Det du skriver inn etter påkalling av kommandoen vil bli sendt som en UDP-pakke til mottakeren. Hvis programmet avsluttes med en feil, er det sannsynligvis forårsaket av at ingen lytter til den gitte porten på den gitte IP-adressen. Sørg for at serveren lytter til riktig port på riktig maskin.

## Utviklingstrinn

En mulig måte å løse denne oppgaven på er i følgende trinn. Vi anbefaler at du setter tapssannsynligheten til 0 før du begynner å teste at tapsgjenopprettingen og gjenoverføringene fungerer.

1. Lag serveren
  - a. Les parametere fra kommandolinjen og sett tapssannsynlighet ved å kalle `set_loss_probability()`.
  - b. Lag datastrukturer for registrerte klienter
  - c. Lag en socket og prøv å sende registrerings- og oppslagsmeldinger ved hjelp av Netcat (bare for å se at mottaket fungerer)
  - d. Implementer registrering. Legg til informasjon i datastrukturen som svar på registreringsmeldinger.
  - e. Implementer oppslag. Svar på oppslag ved hjelp av datastrukturen og informasjonen i forespørselen.
  - f. Test den ved hjelp av print-statements og Netcat for å sende registrerings- og oppslagsmeldinger.
2. Opprett klienten
  - a. Les parametere fra kommandolinjen og sett tapssannsynlighet ved å kalle `set_loss_probability()`.
  - b. Opprett en socket og prøv å sende noe til serveren.
  - c. Implementer registrering.
  - d. Implementer et oppslag av en kjent nick (dvs. en nick som er skrevet rett inn i koden) og lagre informasjonen i en passende datastruktur. Dette er et trinn som skal gjøre det lettere å oppdage feil tidlig.
  - e. Implementer sending og svar på meldinger. Prøv å sende data til en kjent nick ved å bruke informasjonen som er lagret. Implementer et svar på innkommende meldinger.

- f. Implementer parsing av `stdin` slik at du kan erstatte den kjente nicken med en brukergitt nick. Dette vil innebære å gjøre oppslag for å få nødvendig informasjon
    - g. Gjør klienten i stand til å lytte til socket og `stdin` samtidig
  3. Implementer timeout for tapsgjenoppretting for både server og klient. Dette er trinnet når tapssannsynlighet skal settes til en verdi større enn 0.
    - a. Implementer stop-and-wait for registreringsmeldinger. Bruk `select` for å få et tidsavbrudd for gjennforsøk.
    - b. Implementer stop-and-wait for tekstmeldinger. Utvid hovedhendelsesløkken for å få et tidsavbrudd for gjennforsøk.
  4. Implementer hjerteslagmeldinger
    - a. Implementer på serveren og test med Netcat
    - b. Implementer på klienten
  5. Implementer blokkering av sending og mottak fra spesifiserte nicks

## Levering

Du må sende inn all koden din i en enkelt TAR-, TGZ- eller ZIP-arkiv.

Inkluder Makefilen din og inkluder forhåndskoden.

Hvis filen din heter `<kandidatnummer>.tar` eller `<kandidatnummer>.tgz`, vil vi bruke kommandoen `tar` på `login.ifi.uio.no` for å pakke den ut. Hvis filen din heter `<kandidatnummer>.zip`, bruker vi kommandoen `unzip` for å pakke den ut. Sørg for at dette fungerer før du laster opp filen. Det er også fornuftig å laste ned og teste koden etter å ha levert den.

Arkivet ditt må inneholde en Makefile, som skal ha minst disse alternativene:

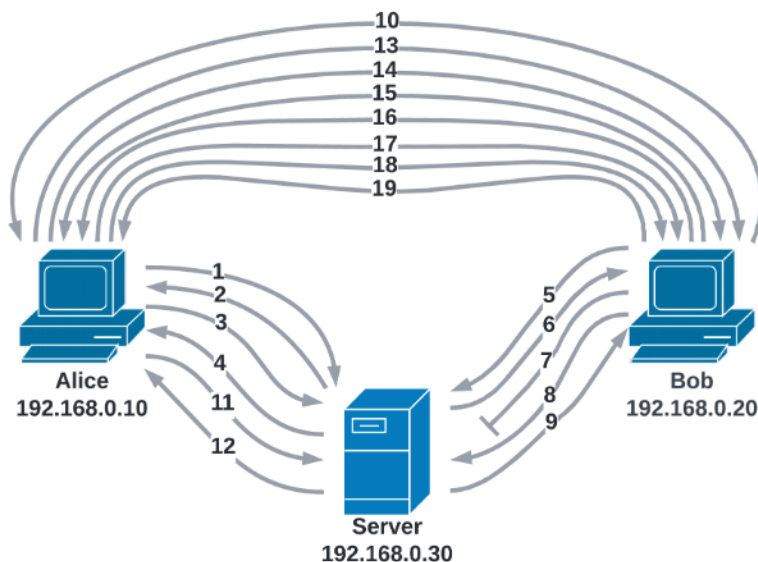
- `make` – kompilerer begge programmene dine til kjørbare binære filer `upush_server` og `upush_client`
- `make all` – gjør det samme som `make` uten noen parameter
- `make clean` – sletter de kjørbare filene og eventuelle midlertidige filer (f.eks. `*.o`)

## Om evalueringen

Hjemmeeksamenen vil bli evaluert på datamaskinene til `login.ifi.uio.no`-poolen. Programmene må kompilere og kjøre på disse datamaskinene. Dersom det er uklarheter i oppgaveteksten bør du påpeke dem og skrive dine valg og forutsetninger i et eget dokument og levere det ved siden av koden.

## Vedlegg: Eksempelscenario

Dette vedlegget viser hvordan et bestemt scenario utspiller seg, som involverer meldingsutveksling mellom to klienter og en server på et lokalnettverk. Først registrerer Alice seg. Hun prøver å slå opp Bob, men han er ennå ikke online. Så registrerer Bob seg og prøver å slå opp Alice. Imidlertid forsvinner hans andre pakke på mystisk vis underveis. Heldigvis slipper den første automatiske gjenoverføringen igjennom. Etter å ha fått Alices adresse, sender Bob henne en melding. Hun er i stand til å svare etter et nytt oppslag – denne gangen med suksess. Til slutt utveksler de et par meldinger. Merk at det er tre par kommuniserende parter, men fire serier med sekvensnumre (for enkelhets skyld starter hver med et unikt siffer i listen nedenfor) siden, i motsetning til TCP, etablerer ikke UPush en forbindelse mellom partene.



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 God morgen, 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 Du og, Bob!
15. [Bob->Alice] ACK 0 OK
16. [Bob->Alice] PKT 1 FROM Bob TO Alice MSG Kaffe eller te?
17. [Alice->Bob] ACK 1 OK
18. [Alice->Bob] PKT 1 FROM Alice TO Bob MSG Kaffe, takk!
19. [Bob->Alice] ACK 1 OK