An emulation of routing and forwarding

For norwegian translation, go to Emulering av ruting og videresending.

Introduction

An important part of networking is in an intermediate node to receive packets and then forward them on an appropriate way towards the destination. The idea in this home exam is to make a program that simulates routing and forwarding in a network where you have to implement a small system that can first set up routing tables and afterwards send messages from one node to all other nodes. For simplicity, we emulate computers using several processes running on the same computer.

Your system must run on IFI's standard Linux machines, and you must use the standard *gcc* compiler installed on these machines. To access such a machine, you can for example run *ssh login.ifi.uio.no*.

Read carefully the section Points at the end of the document. In the section, you can read what is important for points, you get a starting recommendation, and also what is absolutely necessary to get any points at all.

Note that this task is counting in your overall grade. It is therefore not allowed to submit code downloaded or shared from others. All code should be written by yourself. However, note also that this does not forbid collaboration. In contrast, we strongly encourage collaborations where you discuss and share ideas and exchange knowledge, but as stated above: all the submitted code should be your own!

The Task

For the sake of simplicity, the routing and forwarding are emulated by local processes on a single machine and L4 ports are used as unambiguous identifiers - addresses.

We want to emulate a network with N nodes, and we start one process for each of these nodes. Every node can only send packets to directly connected neighbouring nodes in the network. To send to direct neighbours, we use UDP packets.

One of the N nodes (processes) is the sender. This node has always the OwnAddress 1 and is called "node 1" in this text. Node 1 wants to send messages to several other nodes using the shortest path to the destination. Only node 1 starts the transmission of messages, and all other nodes should be able to receive and forward packets.

The shortest path computation is computed by the central routing server - the `routing_server`, which is not part of the forwarding network. Instead, every node has a separate, direct TCP connection with `routing_server`. `routing_server` receives edge weights from all other nodes, computes Dijkstra's algorithm and sends one dedicated routing table to each of the nodes.

Your goal is to collect the connectivity information on `routing_server` from each node, compute the shortest paths from node 1 to all the other nodes and install the routes in the respective nodes' routing table. Then, node 1 should send given messages through the emulated network using UDP packets.

You should write 2 programs called `node` and `routing_server`, one for a node and another one for the central server.

About the realism of this experiment:

- Using a central server to compute all routes inside a network is currently a hot topic, so it is realistic. These networks can get very large. Central route computation is not typical between networks that have different owners.
- Using a direct connection from every node in the network to the central server computing routes is a strong simplification. It is unrealistic.

The emulated network

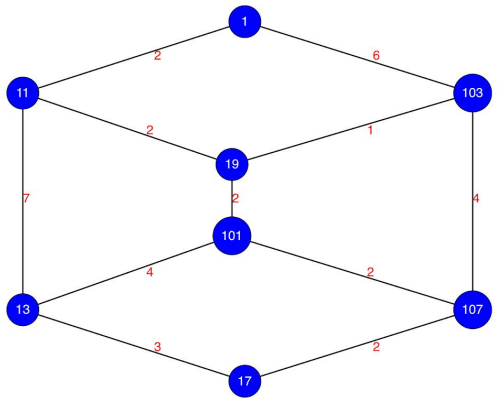

Figure 1: An example of a graph whose edges have weights. The weight is independent of direction.

The nodes are arranged in a graph with weighted bi-directional edges as shown in Figure 1. These edge weights represent the cost of sending a message over this edge. The weights are the same in both directions. Each node knows some of its direct neighbours in the graph, as well as the edge weights to them.

The routing server

The `routing_server` process starts first. It takes as parameters a TCP port P and a number of nodes N. The address of a node will be defined later in the section The nodes. We give some more information about port P in the section The base port.

Usage: `routing_server <P> <N>`
`<P> the TCP port on which the server listens.`
`<N> The number of nodes`

The `routing_server` is a server. It waits for TCP connections from the N nodes, receives their data containing their address and edge information. You must design the protocol for this data transfer, but you should make sure that the server has enough information to detect the address of the node that sends the data, the length of the data, the neighbouring nodes and the weights. Use comments to explain your protocol to us.

When the `routing_server` has received data from all N nodes, it first tests that every edge between two nodes has been reported twice: once by each of the two nodes, and that both nodes reported the same weight. For edges where this is not the case, the `routing_server` prints a warning and ignores the edge.

After this check, the `routing_server` computes all shortest path routes from node 1 to every other node. For this, it uses Dijkstra's Shortest Path First algorithm, using the weights provided by the nodes.

After computing the shortest paths from node 1 to every other node, `routing_server` calls the function

```
void print_weighted_edge( short from_node, short to_node, int pathlen )
```

once for every combination of 2 nodes in the network, including all cases where `from_node == to_node` (25 calls if there are 5 nodes).

The `pathlen` parameter is the path length (sum of all link weights) from node 1 to `from_node`, but only if `from_node` is on the shortest path from node 1 to `to_node`. Otherwise, `pathlen` is -1.

After that, the `routing_server` sends a routing table to every node containing pairs (destination:next hop) without any weights. Next hop is the address of the neighbour. Only those (destination:next hop)-pairs that can be used to reach a destination from node 1 are sent to a node. Some nodes may receive empty routing tables. You design the protocol for this data transfer. Use comments to explain your protocol to us.

The routing server should quit when it has sent one routing table to every client. You should make sure to close all sockets and release all memory.

The nodes

The nodes take as parameters the base port P, their own address A (a number 1-1023), and any number of edges to direct neighbours, where each edge is given as a pair NeighbourAddress:weight - two decimal numbers separated by the character ':'. We explain the idea of the base port P in the section The base port.

Usage: `node <Port> <OwnAddress> <NeighbourAddress>:<weight> ...`

```
<Port>  Value between 1024 and 65535-number of nodes
<OwnAddress> Node's address written as decimal number (1-1023)
<NeighbourAddress>:<weight>
       <NeighbourAddress> - Neighbour's address
       <weight> - Link's weight to corresponding neighbour (an integer number)
```

The node with OwnAddress 1 has a special role. We compute the shortest paths from node 1 to all other nodes, and node 1 is the only node that initiates sending of messages. The node functions that are only used if the node's OwnAddress is one are explained below in The node with OwnAddress 1.

When the node A starts, it tries to bind a UDP socket on port P+A. For example, if the Port parameter of node A has the value P=6000 and the node has the address A=72, then A should be creating a UDP socket with port number P+A=6072.

Before attempting to bind the UDP socket to the port P+A, node A should check that P+A is a valid port number. This socket should be suitable for sending to other nodes and for receiving from other nodes. So, if A wants to send a packet to the node with id B, it has to send the packet to port P+B.

If the node cannot bind the socket to the port, the following should happen:

- the node prints an error message to stderr and quits with error code -1 (see: man exit).

- the user will stop the experiment and try again with a new value for P.

After bind has been performed, node A opens a TCP connection to the `routing_server` and sends the edge information. If the connection fails

- node A quits with error code -2.

After the TCP connection has been established, node A waits for data from the `routing_server` containing its routing table. When node A receives data from `routing_server`

- it may have to read several times to receives all bytes
- it creates an appropriate data structure to store its routing table
- it may close its connection to the `routing_server` after receiving the complete routing table

After receiving a complete routing table, node A waits for packets from other nodes. Only node 1 behaves differently, see below. When node A receives a packet from another node

- memory is managed as appropriate for your code.
- the node receives the entire packet. The packets below explains the format of the packet.
- after that, it checks whether that packet's destination address is its own address.
- if yes,
    - it passes a pointer to the buffer containing the packet to the function

      ```
      void print_received_pkt( short ownAddress, unsigned char* packet )
      ```

    - after this, it inspects the string contained in the message
    - if the message contains only the word QUIT, it closes all sockets, releases all remaining memory, and calls exit with value 0
- else
    - it passes a pointer to the buffer containing the packet to the function

      ```
      void print_forwarded_pkt( short ownAddress, unsigned char* packet )
      ```

    - it uses the routing table provided by routing_server to find the next neighbour on the route towards the destination contained in the message. The message is not changed.
-

The node with OwnAddress 1

This is a normal node that has a special role in this experiment. It has the OwnAddress 1.

This node is the source of the shortest path tree and should read messages from the file "data.txt". No other nodes should read from "data.txt". After receiving its routing table from the routing server, node 1 it waits for 1 second, doing nothing. Then, it reads the human-readable file "data.txt" where every line contains one destination address, followed by one space and then a string. The string ends with the return character '\n', which is not part of the message. For example, in the line

```
814 Good luck with the home exam
```

The string "Good luck with the home exam" is a message that is stored in a packet for sending. It stores also in the packet: the length (35), the source address (1), and the destination address (814). The section "The packets" explains how packets must look.

The node 1 uses its routing table to find the address of the neighbouring node that is on the shortest path from node 1 to node 814. For example, the address of that neighbouring node may be 17. Node 1 will then send the packet to

the node 17. It will do this by sending a UDP packet to the process with the UDP port that is computed by adding base port (6000) and node address (17), that means it will send the packet to UDP port 6017 on localhost.

The file “data.txt” will only contain strings that are shorter than 1000 bytes, and it will be human-readable.

For every line:

- It creates a packet in dynamically allocated heap memory.
- It fills in packet length, destination address, source address and the message (without address D)
- After that, it calls the function

  ```
  void print_pkt( unsigned char* packet )
  ```

  with a pointer to the dynamically allocated buffer.

- After that, it sends the packet to the next node on the route towards the destination D, using the routing table provided by routing_server.
- Memory is released as appropriate for your code.

The packets

A packet is constructed in memory and sent as a single datagram over UDP. It has the following format:

packet length (2 bytes in network byte order)
destination address (2 bytes in network byte order)
source address (2 bytes in network byte order)
message (this a string that must always be 0-terminated)

The packet length is given in bytes and includes the bytes for packet length, addresses, and the message including the 0-character.

Our print functions expect a pointer that refers to memory that contains the whole packet with exactly this structure.

The base port

In the assignment, you are opening several sockets that can wait for events. The TCP socket in `routing_server` is used to wait for connections from nodes, the UDP socket in each node is used to wait for packets from other nodes, as well as for sending packets.

You can choose fixed ports for each process, but if somebody else is using the same computer at the same time, you may not get the number you want. Also, if your program crashed recently, you may not get the same port again without waiting for several minutes. It is therefore convenient if you can very quickly choose new ports for the `routing_server` and all of your nodes.

Our idea is that every process should be started with a base port P. For `routing_server`, this is the TCP port that it uses for listening to connections. All the nodes add their own address to the port number P, and use that number

for their UDP port. For example, if the base port is 6017 and the node address is 44, the node will bind to the UDP port 6061.

If the `bind()` function returns an error for in any of your processes, you can simply stop the experiment, restart it with a new value for P (for example 12752), and all processes will choose a different port than in the previous run. For example, if you choose the new base port 12752, the node with address 44 will try to bind to the UDP port 12796. It might be useful to set the socket option `SO_REUSEADDR` on your sockets before you bind them.

The rest

The file you received from us contains a file print_lib.c, which implements the functions

```
void print_weighted_edge( short from_node, short to_node, int pathlen )
void print_received_pkt( short ownAddress, unsigned char* packet )
void print_forwarded_pkt( short ownAddress, unsigned char* packet )
void print_pkt( unsigned char* packet )
```

You must include the header file print_lib.h, and compile print_lib.c and link it with your own code.

The provided scripts start all processes in the correct order.

Don't forget to set the socket option `SO_REUSEADDR` on the TCP server socket. It helps in debugging.

UDP will never split packets. When you send a packet with UDP, it arrives at the receiver completely or not at all. TCP is different. When you send data with TCP, it can happen that the receiving TCP socket does not read all of these data with a single read() or recv() function call. It will probably not happen here, but you should check if the received number of bytes is identical to the expected number of bytes.

Note that it is possible to solve the assignment without using select or threads in any of the nodes. We have chosen UDP for sending packets between nodes to make this possible. A working TCP solution without select or threads would be extremely difficult.

Submission

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

If your file is called `<candidatenumber>.tar` or `<candidatenumber>.tgz` or `<candidatenumber>.tar.gz`, 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.

Your archive must contain `Makefile` which will have at least these options
`make` - compiles both your programs resulting in executable binaries "node" and "routing_server"
`make all` - does the same as make without any parameter
`make clean` - deletes the executables and any temporary files (eg. *.o)
`make run` - execute both of the scripts that we provide

Your archive must also contain your code, our code and our scripts.

Points

You will get full points if:

- Makefile works with make, make run and make clean as expected, and
- Output is identical to that of a flawless working solution
    - for 2 example networks provided by us,
    - and for 2 additional example networks of similar size that we are not sharing, and
- Code runs on login.ifi.uio.no with valgrind showing no memory leaks and no other memory errors, and
- Code implements the requested solution

You will get no points (failing the exam) if:

- Delivery cannot be extracted with either of the commands tar or zip on login.ifi.uio.no, or
- Delivery does not contain Makefile and source code files, or
- Calling “make” does not compile the code on login.ifi.uio.no, or
- Code does not run on login.ifi.uio.no with any of the 2 example networks provided by us.

Otherwise, we will check

- Makefile
- Files
    - Correct reading of messages from “data.txt”
    - Correct creation of packets from messages, including correct length and addresses
- Memory
    - Memory leaks
    - Correct handling of strings including 0-termination (allocating ridiculously much space for a string is OK)
- TCP connections
    - Code for making TCP connections is correct
    - Waiting for several connections is correct
    - Mapping from socket to node address is correct in routing server, allowing it to answer to the right node
- UDP messages
    - Mapping from receiving socket to node address is correct in nodes
    - Mapping from node address to sending socket is correct in nodes
- Route computation and distribution
    - routing_server uses a suitable data structure for collecting neighbour information from all nodes
    - Shortest path routes from node 1 to all other nodes are computed according to Dijkstra’s algorithm
    - Route computation is correct (even if the extracted routing tables are wrong)
    - A correct routing table is computed for every node (even if it is sent to the wrong node)
- Packet sending
    - Packets are routed correctly
    - Packets are implemented in the required format, including network byte order and 0-termination
- All functions are integrated correctly.

Recommendations

It is strongly recommended that you develop your solution for Dijkstra’s algorithm and routing table creation first without the TCP server functionality. You could read the graphs of the examples that we provide from file, or you

could fill them into hand-written data structures in your code. If the complete program does not work, you could still get full points for Dijkstra' algorithm if this is done correctly.

It is similarly recommended to implement nodes first with routing tables that loaded from file or hand-written in the code. You can then implement parts of the assignment such as reading of messages from "data.txt", UDP sending to a destination, receiving and forwarding of packets. If done correctly, this could already give full points for "correct handling of strings", "mapping from socket to node", "mapping from node to socket", and "packets are in the required format".

If you deliver one program for Dijkstra's algorithm and one program for nodes, but have not connected them, you will receive points for both parts.

Emulering av ruting og videresending

Introduksjon

En viktig del av et nettverk er at en intermediate node mottar pakker og deretter videresender pakkene på en hensiktsmessig måte mot målet. Ideen i denne hjemmeeksamen er å lage et program som simulerer ruting og videresending i et nettverk, der du må implementere et lite system som først kan sette opp rutingtabeller og deretter sende meldinger fra en node til alle andre noder. For enkelhet skyld skal dere simulere nettverksnoder ved hjelp av flere prosesser som kjører på samme datamaskin.

Systemet ditt må kjøre på IFI sine standard Linux-maskiner, og du må bruke standard gcc-kompilatoren som er installert på disse maskinene. For å få tilgang til en slik maskin, kan du for eksempel kjøre *ssh login.ifi.uio.no*.

Les nøye avsnittet Poeng på slutten av dokumentet. I det avsnittet kan du lese hva som er viktig for å få poeng, og du får en anbefaling om hvordan du bør starte. Du kan i tillegg se hva som er absolutt nødvendig for å få noen poeng i det hele tatt.

Merk at denne oppgaven teller som en del av din samlede karakter i kurset. Det er derfor ikke tillatt å sende inn kode som er lastet ned eller delt fra andre. All kode skal skrives av deg selv. Vær imidlertid oppmerksom på at dette ikke forbyr samarbeid. Vi oppfordrer sterkt til samarbeid hvor du diskuterer og deler ideer og utveksler kunnskap, men som nevnt ovenfor: all innsendt kode skal være din egen!

Oppgaven

For enkelhets skyld er rutingen og videresending emulert av lokale prosesser på en enkelt maskin, og L4-porter brukes som entydige identifiseringsadresser.

Vi ønsker å etterligne et nettverk med N noder, og vi starter en prosess for hver av disse nodene. Hver node kan bare sende pakker til direkte tilkoblede nabonoder. For å sende til direkte naboer bruker vi UDP-pakker.

En av de N nodene (prosessene) er avsender. Denne noden har alltid OwnAddress 1 og kalles "node 1" i denne teksten. Når node 1 ønsker å sende meldinger til andre noder skal den bruke den korteste veien (stien (“path”)) gjennom nettverket til destinasjonen. Kun node 1 starter overføringen av meldinger, og alle andre noder skal kunne motta og videresende pakker.

Den korteste veien beregnes av den sentrale rutingserveren (`routing_server`) som ikke er en del av videresendingsnettverket. I stedet har hver node en separat, direkte TCP-forbindelse til `routing_server`. `routing_server` skal motta kant-vekter fra alle andre noder, og så beregne Dijkstras algoritme og sende en dedikert rutingtabell til hver av nodene.

Målet ditt er å samle tilkoblingsinformasjonen på `routing_server` fra hver node, beregne de korteste veiene fra node 1 til alle de andre noder og installere rutene i de respektive rutingtabellene i de forskjellige nodene. Så skal node 1 sende de gitte meldingene gjennom det emulerte nettverket ved hjelp av UDP-pakker.

For å løse oppgaven bør du skrive 2 programmer kalt `node` og `routing_server`, et program for nodene og et annet program for den sentrale serveren.

Sammenlignet med ekte store nettverk har vi gjort noen forenklinger. Realismen i denne oppgaven er derfor som følger:

- Å bruke en sentral server til å beregne alle ruter inne i et nettverk er i dag et aktuelt diskusjonstema, så det er realistisk. Disse nettverkene kan bli svært store. Slik sentral ruteberegning er ikke typisk mellom nettverk som har forskjellige eiere.
- Å bruke en direkte tilkobling fra hver node i nettverket til den sentrale serveren som beregninger rutene er en sterk forenkling. Det er urealistisk.

Det emulerte nettverket

Nodene er arrangert i en graf med toveis-kanter med vekter som for eksempel vist i figur 1. Disse vektene representerer kostnaden ved å sende en melding over denne kanten. Vektene er de samme i begge retninger. Hver node kjenner noen av sine direkte naboer i grafen, og vekten til kanten til dem.

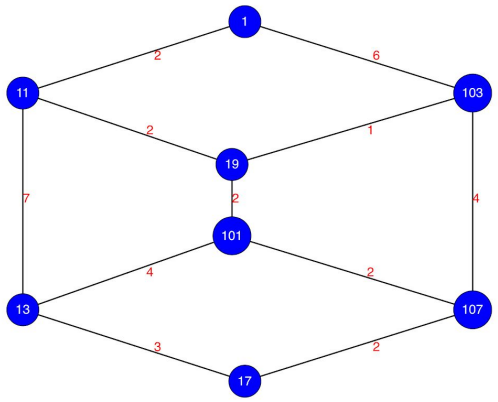

Figure 1: Et eksempel av en graf med. Kanter med vekt, uavhengig av retning

Rutingserveren

Ruting-serverprosessen (`routing_server`) starter først. Den tar som parametere en TCP-port P og et antall noder N. Adressen til en node vil bli definert senere i avsnittet Nodene. Vi gir litt mer informasjon om port P i avsnitt Baseporten.

Bruk: `routing_server <P> <N>`
`<P> TCP porten som serveren lytter til`
`<N> Antall noder`

Ruting-serveren (`routing_server`) er en server. Den venter på TCP-tilkoblinger fra de N nodene, og skal motta dataene som inneholder adresse- og kant-informasjon. Du skal lage protokollen for denne dataoverføringen, og du bør sørge for at serveren har nok informasjon til å oppdage adressen til noden som sender dataene, lengden på dataene, de nærliggende noder og vekter. Bruk kommentarer for å forklare protokollen din til oss.

Når ruting-serveren (`routing_server`) har mottatt data fra alle de N nodene, tester den først at hver kant mellom to noder er blitt rapportert to ganger: en gang for hver av de to noder, og at begge nodene rapporterte samme vekt. For kanter der dette ikke er tilfelle, skal `routing_server` skrive en advarsel og ignorere deretter kanten.

Etter denne sjekken beregner `ruting_server` alle korteste veier fra node 1 til hver annen node. For dette bruker den Dijkstra's Shortest Path First algoritme, ved hjelp av vektene levert av nodene.

Etter å ha beregnet de korteste veiene fra node 1 til alle de andre nodene, kaller `routing_server` funksjonen

```
void print_weighted_edge( short from_node, short to_node, int pathlen )
```

en gang for hver kombinasjon av 2 noder i nettverket, inkludert alle tilfeller der `from_node == to_node` (25 kall hvis det er 5 noder).

Parameteren `pathlen` er vei-lengden (summen av alle vekter) fra node 1 til noden med adresse `from_node`, men bare hvis noden med adresse `from_node` er på korteste sti fra node 1 til `to_node`. Hvis ikke er `pathlen` lik -1.

Etter dette sender ruting-serveren (`routing_server`) en rutingtabell til hver node som inneholder par av (destinasjon:neste hopp) uten noen vekter. Neste hopp er adressen til naboen. Bare de (destinasjon:neste hopp)-parene som kan brukes til å nå et mål fra node 1, sendes til en node. Noen noder kan motta tomme rutingtabeller. Du utformer protokollen for denne dataoverføringen. Bruk kommentarer for å forklare protokollen din til oss.

Rutingserveren (`routing_server`) skal avslutte når den har sendt en rutingtabell til hver klient, og når du avslutter må du passe på å lukke alle socket'er og frigjøre alt minne.

Nodene

Nodene tar som parametre baseporten P, sin egen adresse A (et tall 1-1023), og et hvilket som helst antall kanter til direkte naboer. Hver kant er gitt som et par NeighbourAddress:vekt - to desimaltall separert av tegnet ':'. Vi forklarer ideen om baseporten P i avsnitt Baseporten.

Bruk: `node <Port> <OwnAddress> <NeighbourAddress>:<weight> ...`

```
<Port>  Verdi mellom 1024 og 65535-antall noder
<OwnAddress> Nodens adresse skrevet som desimaltall (1-1023)
<NeighbourAddress>:<weight>
        <NeighbourAddress> - Naboens adresse
        <weight> - En link sin vekt til korresponderende nabo (et tall)
```

Noden med OwnAddress 1 har en spesiell rolle. Vi beregner de korteste banene fra node 1 til alle andre noder, og node 1 er den eneste noden som initierer sending av meldinger. Node-funksjonene som bare brukes hvis nodens OwnAddress er en, er forklart nedenfor i Noden med OwnAddress 1.

Når noden A starter, forsøker den å binde en UDP-socket på port P + A. For eksempel, hvis port-parameteren i node A har verdien P = 6000, og noden har adressen A = 72 så skal node A lage en UDP-socket med portnummer P + A = 6072.

Før du prøver å binde UDP-socketen til porten P + A, bør node A kontrollere at P + A er et gyldig portnummer. Denne socket'en skal kunne brukes både til sending av pakker til andre noder og for mottak fra andre noder. Så, hvis node A ønsker å sende en pakke til node B, må den sende pakken til port P + B.

Hvis noden ikke kan binde socket'en til porten skal

- noden skrive ut en feilmelding til stderr og avslutte med feilkode -1 (se: man exit).
- brukeren stoppe eksperimentet og prøve igjen med en ny verdi for P.

Etter at bind er utført skal node A åpne en TCP-tilkobling til `routing_server` og sende kant-informasjonen den har. Hvis tilkoblingen mislykkes skal

- node A avsluttes med feilkode -2.

Etter at TCP-tilkoblingen er gjort skal node A vente på data fra `routing_server` som inneholder rutingtabellen. Når node A mottar data fra `routing_server`

- må den kanskje leses flere ganger fra socketen for å motta alle bytes
- skaper den en passende datastruktur for å lagre rutingtabellen
- kan den lukke forbindelsen til routing_serveren etter å ha mottatt det komplette rutingtabellen

Etter å ha mottatt en komplett rutingtabell skal node A vente på pakker fra andre noder. Kun node 1 oppfører seg annerledes, se nedenfor. Når node A mottar en pakke fra en annen node skal noden

- håndtere minnet på en god måte i koden.
- sjekke om hele pakken er mottatt. Avsnittet Pakkene nedenfor forklarer formatet til pakken.
- sjekke om den pakkenes destinasjonsadresse er noden sin egen adresse.
- hvis ja, skal noden
    - sende en peker til bufferet som inneholder pakken til funksjonen
      `void print_received_pkt (short ownAddress, unsigned char* packet)`
    - inspisere strengen som finnes i meldingen
    - hvis meldingen bare inneholder ordet QUIT, lukkee alle socketer, frigi alt gjenværende minne, og kalle exit med verdi 0
- ellers skal noden
    - sende en peker til bufferet som inneholder pakken til funksjonen
      `void print_forwarded_pkt (short ownAddress, unsigned char* packet)`
    - bruke rutingtabellen som leveres av `routing_server` for å finne den neste naboen på ruten mot destinasjonen som finnes i meldingen. Meldingen skal ikke endres.

Noden med OwnAddress 1

Dette er en vanlig node som har en spesiell rolle i dette eksperimentet. Den har OwnAddress 1.

Denne noden er kilden til de korteste veiene og skal lese meldinger fra filen "data.txt". Ingen andre noder skal lese fra "data.txt". Etter å ha mottatt sin rutingtabell fra `routing_server`, skal node 1 vente i et sekund uten å gjøre noe. Deretter skal den lese filen "data.txt" der hver linje inneholder en destinasjonsadresse, etterfulgt av ett mellomrom og en streng. Strengen skal slutte med '\n', som ikke er en del av meldingen. For eksempel i linjen

```
814 Good luck with the home exam
```

er strengen "Good luck with the home exam" en melding som er lagret i en pakke for sending. Den lagrer også i pakken: lengden (35), kildeadressen (1) og destinasjonsadressen (814). Avsnittet Pakkene forklarer hvordan pakker må se ut.

Node 1 skal bruke sin rutingtabell for å finne adressen til nabo-noden som er på den korteste ruten fra node 1 til node 814. For eksempel kan adressen node 1 sin nabonode som ligger på den korteste veien til 814 være 17. Node 1 vil da sende pakken til node 17. Den vil gjøre dette ved å sende en UDP-pakke til prosessen med UDP-porten som beregnes ved å legge til base port (6000) og node-adresse (17), som betyr at den vil sende pakken til UDP-port 6017 på localhost.

Filen "data.txt" vil bare inneholde strenger som er kortere enn 1000 byte, og de vil være lesbare med en teksteditor.

For hver linje skal noden:

- lage en pakke i dynamisk allokert heap-minne.
- fylle inn pakkelengde, destinasjonsadresse, kildeadresse og meldingen (uten adresse D)
- kalle funksjonen
  `void print_pkt(unsigned char* pkt)`
  med en peker til den dynamisk allokerte bufferen.
- sende pakken til neste node på ruten mot destinasjonen D, ved hjelp av rutingtabellen som leveres av routing_server.
- frigi minne på en god måte i koden din

Pakkene

En pakke opprettes i minnet og sendes som et enkelt datagram over UDP. Den skal ha følgende format:

pakkelengde (2 bytes i network byte order)
destinasjonsadresse (2 bytes i network byte order)
kildeadresse (2 bytes i network byte order)
melding (dette er en streng alltid må avsluttes med et 0-byte)

Pakkelengden er gitt i antall bytes og inneholder bytes for pakkelengde, adresser, og meldingen inkludert 0-tegnet.

Merk at våre utskriftsfunksjoner forventer en peker som refererer til minne som inneholder hele pakken med nøyaktig denne strukturen.

Baseporten

I denne oppgaven skal du åpne flere sockets som kan vente på hendelser. TCP-socketen i `routing_server` brukes til å vente på tilkoblinger fra noder, UDP-socketen i hver node brukes til å vente på pakker fra andre noder, så vel som for sending av pakker.

Du kan velge faste porter for hver prosess, men hvis noen andre bruker samme datamaskin samtidig, kan du ikke få nummeret du vil ha. I tillegg, hvis programmet krasjer kan du ikke få samme port igjen uten å vente flere minutter. Det er derfor praktisk hvis du raskt kan velge nye porter for `routing_server`'en og alle dine noder.

Vår idé er at hver prosess skal startes med en baseport P. For `routing_server` er dette TCP-porten som den bruker til å lytte på for tilkoblinger. Alle noder skal legge til sin egen adresse til portnummeret P, og bruke det nummeret til UDP-porten. For eksempel, hvis baseporten er 6017 og noden adressen er 44, vil noden binde til UDP-porten 6061.

Hvis funksjonen `bind()` returnerer en feil for noen av prosessene dine, kan du bare stoppe eksperimentet, starte det på nytt med en ny verdi for P (for eksempel 12752), og alle prosesser skal da velge en annen port enn ved forrige kjøring. Hvis du for eksempel velger den nye baseporten 12752, vil noden med adresse 44 forsøke å binde til UDP-porten 12796. Det kan være nyttig å sette socket option `SO_REUSEADDR` på socketene dine før du binder dem.

Resten

Filen du mottok fra oss inneholder en fil *print_lib.c*, som implementerer funksjonene
`void print_weighted_edge( short from_node, short to_node, int pathlen )`

```
void print_received_pkt( short ownAddress, unsigned char* packet )
void print_forwarded_pkt( short ownAddress, unsigned char* packet )
void print_pkt( unsigned char* packet )
```
Denne filen skal ikke forandres, og du må inkludere headerfilen print_lib.h, og kompilere print_lib.c og linke den med din egen kode.

De gitte skriptene starter alle prosessene i riktig rekkefølge.

Ikke glem å sette socket option `SO_REUSEADDR` på TCP-server-socketen. Det hjelper med feilsøking.

UDP vil aldri dele opp pakker. Når du sender en pakke med UDP, kommer den til mottakeren i sin helhet eller ikke i det hele tatt. TCP er annerledes. Når du sender data med TCP kan det hende at TCP-socketen hos mottaker ikke leser alle disse dataene med et enkelt funksjonskall til `read()` eller `recv()`. Det vil nok ikke skje her, men du bør sjekke om antall mottatte bytes er identisk med det forventede antall bytes.

Merk at det er mulig å løse oppgaven uten å bruke select eller tråder i noen av nodene. Vi har valgt UDP for å sende pakker mellom noder for å gjøre dette mulig. En fungerende TCP-løsning uten select eller tråder ville være ekstremt vanskelig.

Innlevering

Du må pakke og sende inn all din kode i et enkelt TAR, TAR.GZ eller ZIP-arkiv.

Hvis filen din heter `<kandidatnummer>.tar`, `<kandidatnummer>.tgz` eller `<kandidatnummer>.tar.gz`, vil vi bruke kommandoen `tar` på login.ifi.uio.no for å pakke ut den. Hvis filen din heter `<kandidatnummer>.zip`, vil vi bruke kommandoen `unzip` på login.ifi.uio.no for å pakke den ut. Pass på at dette fungerer før du laster opp filen.

Arkivet ditt må inneholde en `Makefile` som må minst ha disse alternativene
`make` - kompilerer begge programmene dine til eksekverbare filer "node" and "routing_server"
`make all` - gjør det samme som make uten noen parametere
`make clean` - sletter de eksekverbare og midlertidige filer (f.eks *.o)
`make run` - kjører begge skriptene som vi har gitt

Arkivet ditt må også inneholde både din og vår kode samt våre skript.

Poeng

Du får maksimalt antall poeng hvis:

- Makefile fungerer med make, make run og make clean fungerer som forventet, og
- Output er identisk med en feilfri løsning
    - for de 2 eksempelnettverkene levert av oss, og
    - for 2 ekstra eksempelnettverk av samme størrelse som vi *ikke* deler ut, og
- Koden kjører på login.ifi.uio.no med valgrind som ikke viser noen minnelekkasjer eller andreminnefeil, og
- Koden implementerer den forespurte løsningen

Du får ingen poeng (stryker eksamen) hvis:

- Leveringen ikke kan pakkes ut med hverken tar eller zip på login.ifi.uio.no, eller
- Leveringen ikke inneholder Makefile og kildekodefiler, eller

- Det å kjøre "make" ikke kompilerer koden på login.ifi.uio.no eller
- Koden kjører ikke på login.ifi.uio.no med noen av de 2 eksemplene som er gitt av oss.

Ellers vil vi sjekke

- Makefile
- Filer
    - Riktig innlesing av meldinger fra “data.txt”
    - Riktig opprettelse av pakker fra meldinger, inkludert riktig lengde og adresser
- Minne
    - Minnelekkasjer
    - Korrekt håndtering av strenger inkludert 0-avslutning (tildeling av latterlig mye plass til en streng er OK)
- TCP-tilkoblingen
    - Koden for å lage TCP-tilkoblinger er riktig
    - Programmet venter på flere tilkoblinger på riktig vis
    - Mapping fra socket til node er korrekt i rutingserveren, slik at den kan svare til rett node
- UDP meldinger
    - Mapping fra mottaker-socket til nodeadresse er korrekt i nodene
    - Mapping fra nodeaddresse til sende-socket er korrekt i nodene
- Ruteberegning og distribusjon
    - `routing_server` bruker en passende datastruktur for å samle naboinformasjon fra alle noder
    - Korteste rute-vei fra node 1 til alle andre noder beregnes i henhold til Dijkstras algoritme
    - Ruteberegning er riktig (selv om de utviste rutingtabellene er feil)
    - En riktig rutingtabell beregnes for hver node (selv om den sendes til feil node)
- Pakkesending
    - Pakker er rutet riktig
    - Pakker er implementert i ønsket format, inkludert network byte order og 0-avslutning
- Alle funksjoner er integrert riktig.

Anbefalinger

Det anbefales sterkt at du utvikler løsningen for Dijkstras algoritme og oppretter rutingtabellen først uten TCP-server funksjonaliteten. Du kan lese grafer av eksemplene vi tilbyr fra fil, eller du kan fylle dem i håndskrevne datastrukturer i koden din. Hvis hele programmet ikke virker, kan du fortsatt få full poeng for Dijkstras algoritme hvis dette er gjort riktig.

Det anbefales på samme måte å implementere noder først med rutingtabeller som lastes fra fil eller håndskrevet i koden. Du kan da implementere deler av oppgaven som lesing av meldinger fra "data.txt", at UDP sender til en destinasjon, og at mottak og videresending av pakker. Hvis det gjøres riktig, kan dette gi full poengscore for "riktig håndtering av strenger", "mapping fra socket til node", "mapping fra node til socket" og "pakker i ønsket format".

Leverer du et programm for Dijkstras algoritme og et for nodene, men de er ikke er knyttet sammen, vil du motta poeng for begge deler.