Datakommunikasjon og Nettverksprogrammering, Hovedrapport

Odd-Erik Frantzen, gruppe 34

Innhold

[1. Introduksjon 3](#_Toc6181060)

[1.1 Introduksjon 3](#_Toc6181061)

[1.2 Felles begreper 3](#_Toc6181062)

[1.3 Kompilering med Boost 3](#_Toc6181063)

[2. Øving 3 – Kommunikasjon over TCP. Matteklient og ensides webserver. 3](#_Toc6181064)

[2.1 Begreper 3](#_Toc6181065)

[2.2 Programmeringsmetoder 3](#_Toc6181066)

[2.3 Utdrag fra kode 4](#_Toc6181067)

[2.3.1 Server 4](#_Toc6181068)

[2.3.2 Klient 5](#_Toc6181069)

[2.3.3 Web server 7](#_Toc6181070)

[2.4 Pakkefangst 7](#_Toc6181071)

[2.4.1 Matteklient og Matteserver 7](#_Toc6181072)

[2.4.2 Web server 8](#_Toc6181073)

[2.5 Diskusjon 9](#_Toc6181074)

[3. Øving 4 – Kommunikasjon over UDP 10](#_Toc6181075)

[3.1 Begreper 10](#_Toc6181076)

[3.2 Programmeringsmetoder 10](#_Toc6181077)

[3.3 Utdrag fra kode 10](#_Toc6181078)

[3.3.1 Server 10](#_Toc6181079)

[3.3.2 Klient 12](#_Toc6181080)

[3.4 Pakkefangst 13](#_Toc6181081)

[3.5 Diskusjon 13](#_Toc6181082)

[4. Øving 5 – Programmatisk bruk av databaser via ORM 14](#_Toc6181083)

[4.1 Begreper 14](#_Toc6181084)

[4.2 Programmeringsmetoder 14](#_Toc6181085)

[4.2.1 Kompilering 14](#_Toc6181086)

[4.2.2 ORM objekter i ODB 14](#_Toc6181087)

[4.2.3 Objekter/Klasser/Metoder brukt: 15](#_Toc6181088)

[4.3 Utdrag fra kode 15](#_Toc6181089)

[4.3.1 Definisjon av Account tabellen i C++ 15](#_Toc6181090)

[4.3.2 Oppgave 2 – skapelse, lasting og persistence 16](#_Toc6181091)

[4.3.3 Oppgave 3 og 4 – Race condition skrivefeil, og optimistisk låsing 17](#_Toc6181092)

[4.4 Pakkefangst og MySQL verdier under kjøring 18](#_Toc6181093)

[4.4.1 Oppgave 2 18](#_Toc6181094)

[4.4.2 Oppgave 3 20](#_Toc6181095)

[4.4.3 Oppgave 4 20](#_Toc6181096)

[4.5 Diskusjon 21](#_Toc6181097)

[5. Øving 6 – Websocket 22](#_Toc6181098)

[5.1 Begreper 22](#_Toc6181099)

[5.2 Programmeringsmetoder 22](#_Toc6181100)

[5.3 Utdrag fra kode 22](#_Toc6181101)

[5.4 Pakkefangst 26](#_Toc6181102)

[5.5 Diskusjon 27](#_Toc6181103)

# 1. Introduksjon

## 1.1 Introduksjon

Full kildekode for alle øvingene ligger ute på <https://github.com/odderikf/datacom>,

## 1.2 Felles begreper

C++ - valgt programmeringsspråk for de fleste øvinger.

Boost – en gruppe biblioteker til C++ ment til å utvide standardbiblioteket, altså “booste” det.

I/O - In/Out, å ta inn data fra drivere og gi data ut til drivere, deriblant nettverkskort.

ASIO - Står for Async I/O. Det er et bibliotek under Boost som tilbyr funksjoner for synkrone og asynkrone I/O operasjoner, slik som TCP og UDP

## 1.3 Kompilering med Boost

Asio importeres da som <boost/asio.hpp>.

For å kompilere må det da legges til linkerflagg. I CMake kan dette gjøres med

find\_package(Boost 1.65.1 REQUIRED COMPONENTS system)  
include\_directories(${Boost\_INCLUDE\_DIR})  
target\_link\_libraries(TARGET ${Boost\_LIBRARIES})

# 2. Øving 3 – Kommunikasjon over TCP. Matteklient og ensides webserver.

## 2.1 Begreper

C++ - valgt programmeringsspråk for øvingen.

ASIO – Biblioteket som har blitt brukt for asynkron TCP I øvingen.

TCP – Protokoll på transportlaget som brukes til bl.a. HTTP

## 2.2 Programmeringsmetoder

Jeg har valgt å bruke C++ som programmeringsspråk, og biblioteket ASIO som er del av Boost (Boost::ASIO).

Namespacet boost::asio::ip::tcp er mye brukt, og det er derfor “using boost::asio::ip::tcp ” I toppen av filene. Da slipper jeg altså å skrive fullt namespace hver gang namespacet tcp er brukt.

Det brukes også shared pointers, disse er I <memory> modulen I C++ sitt standardbibliotek. Disse er da pointers (referanser til hvor I minne noe er) med minnesikkerhet lagt oppå seg.

Generelt brukes:

boost::asio::io\_service  
tcp::socket  
std::make\_shared<T>(CONSTRUCTOR ARGS)  
std::shared\_ptr<T>   
std::getline(istream, string)  
std::cout  
boost::asio::streambuf  
boost::asio::async\_read\_until(socket, buffer, markerstring, callback)  
boost::asio::async\_write(socket, buffer, callback)  
std::string  
std::istream, std::ostream  
std::exception

For matteserver og webserver er det også:  
tcp::endpoint;  
tcp::acceptor;

Mens for matteklient:  
tcp::resolver

## 2.3 Utdrag fra kode

### 2.3.1 Server

Startpunkt, lager et serverobjekt og starter det:

int main() {  
  
 WebServer server;  
 server.start();  
  
 return 0;  
}

Konstruktør lager et endepunkt og en akseptør til å ta imot requests:

WebServer() : endpoint(tcp::v4(), 8081), acceptor(io\_service, endpoint){}

Start starter en accept\_request, som vil starte en asynk prosess som tar seg av requests, og så starter den io\_service sin mainloop:

void start(){  
 accept\_request();  
 Io\_service.run();  
}

Accept\_request aksepterer nye requests og gir de en TCP socket. Deretter kjører den en ny instans av seg selv asynkront når den mottar en request, og gir socket til handle\_request loopen

void accept\_request(){  
 auto socket = std::make\_shared<tcp::socket>(io\_service);  
 acceptor.async\_accept(\*socket, [this, socket](const boost::system::error\_code &ec){  
 accept\_request();  
 if(!ec){  
 std::cout << "boop" << std::endl;  
 handle\_request(socket)  
 }  
  
 });  
}

Handle\_request tar seg av en klientforbindelse, ved å lese data fra request, tolke den, og sende et svar:

void handle\_request(const std::shared\_ptr<tcp::socket> &socket){  
 auto read\_buffer = std::make\_shared<boost::asio::streambuf>();  
 boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n", [=](const boost::system::error\_code &ec, size\_t){  
 if(!ec){  
 std::string message;  
 std::string failmessage;  
 std::istream read\_stream(read\_buffer.get());  
 std::getline(read\_stream, message);  
 message.pop\_back();  
 if(message == "exit") return;

[ Strengtolkning og så videre for kalkulator ]

auto write\_buffer = std::make\_shared<boost::asio::streambuf>();  
 std::ostream write\_stream(write\_buffer.get());  
 if(failmessage.length()) write\_stream << failmessage << "\r\n";  
 else write\_stream << (isPlus ? a+b : a-b) << "\r\n";  
  
 // Write to client  
 boost::asio::async\_write(\*socket, \*write\_buffer, [this, socket, write\_buffer](const boost::system::error\_code &ec, size\_t) {  
 // If not error:  
 if (!ec) {  
 std::cout << "server: reply sent to client" << std::endl;  
 handle\_request(socket);  
 }  
 });  
  
 }  
 });  
}

### 2.3.2 Klient

Entry point starter en MathClient mot localhost:8081:

int main() {  
 MathClient("127.0.0.1", 8081);  
}

Konstruktør tar en url og en port, og resolver asynkront hvor serveren er. Den kjører så io\_service sin mainloop. Inni callbacken lager den en socket, kobler seg mot serveren, og går over til matteklient-loopen ask:

MathClient(const std::string &host, unsigned short port) : resolver(io\_service) {  
 // Create query from host and port  
 auto query = tcp::resolver::query(host, std::to\_string(port));  
 // Resolve query (DNS-lookup if needed)  
 resolver.async\_resolve(query, [this](const boost::system::error\_code &ec,  
 boost::asio::ip::tcp::resolver::iterator it) {  
 // If not error:  
 if (!ec) {  
 std::cout << "client: query resolved" << std::endl;  
  
 auto socket = std::make\_shared<tcp::socket>(io\_service);  
  
 boost::asio::async\_connect(\*socket, it, [this, socket](const boost::system::error\_code &ec,  
 boost::asio::ip::tcp::resolver::iterator /\*it\*/) {  
 // If not error:  
 if (!ec) {  
  
 std::cout << "client: connected to server" << std::endl;  
 std::cout << "Syntaks: a+b eller a-b" << std::endl;  
 ask(socket);  
  
 }  
 });  
 }  
 });  
  
 io\_service.run();  
}

Ask-loopen skriver da meldingen til server, leser svaret, printer det ut, og rekurserer (altså looper):

void ask(std::shared\_ptr<tcp::socket> const& socket){  
 // write buffer with automatic memory management through reference counting  
 auto write\_buffer = std::make\_shared<boost::asio::streambuf>();  
 std::ostream write\_stream(write\_buffer.get());  
 std::string message;  
  
 std::getline(std::cin, message);  
 write\_stream << message << "\r\n"  
 "";  
 if(message == "exit") return;  
  
 boost::asio::async\_write(\*socket, \*write\_buffer, [this, socket,   
 write\_buffer](const boost::system::error\_code &ec, size\_t) {  
 // If not error:  
 if (!ec) {  
 auto read\_buffer = std::make\_shared<boost::asio::streambuf>();  
 // Read from client until newline ("\r\n")  
 boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n", [this,   
 socket, read\_buffer](const boost::system::error\_code &ec, size\_t) {  
 // If not error:  
 if (!ec) {  
 std::string message;  
 std::istream read\_stream(read\_buffer.get());  
 std::getline(read\_stream, message);  
 std::cout << message << std::endl;  
 ask(socket);  
 }  
 });  
 }  
 });  
}

### 2.3.3 Web server

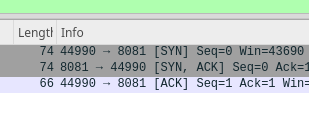
Her er det egentlig mye av det samme som i matteserver, hovedforskjellen er meldingen som blir sendt i svar:

std::string response = "HTTP/1.1 200 OK\r\nconnection: keep-alive\r\n\r\n<html>\r\n<body>\r\n";  
response += "<h1>Velkommen</h1>\r\n";  
response += "<ul>";  
while(std::getline(read\_stream, message)){  
 message.pop\_back();  
 if(message != "")  
 response += ("<li>" + message + "</li>\r\n");  
}  
response += "</ul>\r\n";  
response += "</body>\r\n</html>\r\n\r\n";  
write\_stream << response;

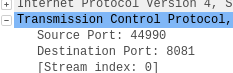
## 2.4 Pakkefangst

### 2.4.1 Matteklient og Matteserver

Starter server og klient, og fanger da kommunikasjonen mellom dem. Ser at det skjer en 3-way handshake med syn, syn/ack, ack

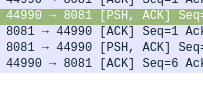


Ser da på første melding sitt TCP felt og ser at melding sendes fra en dynamisk port (da klient ikke oppgir port) til port 8081 der serveren lytter.

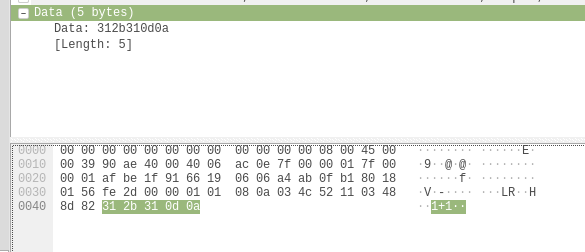


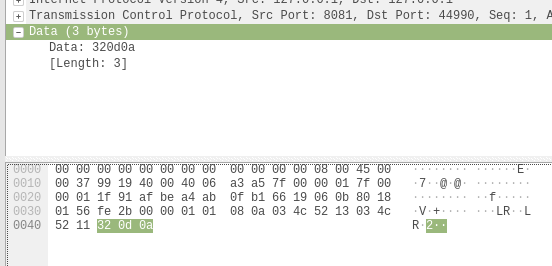
Serveren vet da at den nye socketen skal koble til port 44990

Sender så «2+2» fra klienten, og ser da at klienten sender hele meldingen i èn TCP request, som da har PSH for å vise at meldingen er ferdig. Det sendes også med ACK i hver melding, i tillegg til at det noen ganger kommer separate ACK. Serveren svarer da ACK, og så PSH,ACK med svaret på 2+2, som da er 4.



Her kan vi da se dataen i klartekst:



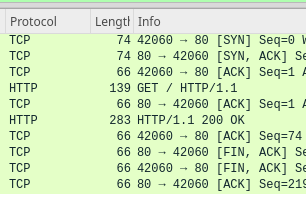


Til slutt skriver jeg exit i klienten, som da returnerer uten å sende melding. Når socketen da går ut av scope, så blir den destruert (selv om den er pointer, da smart pointers har referansetelling). Da sendes FIN til server, og server svarer FIN.

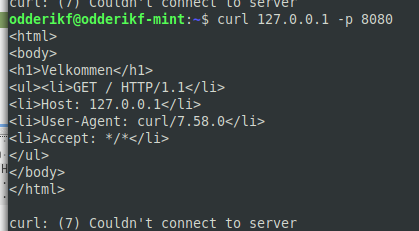


### 2.4.2 Web server

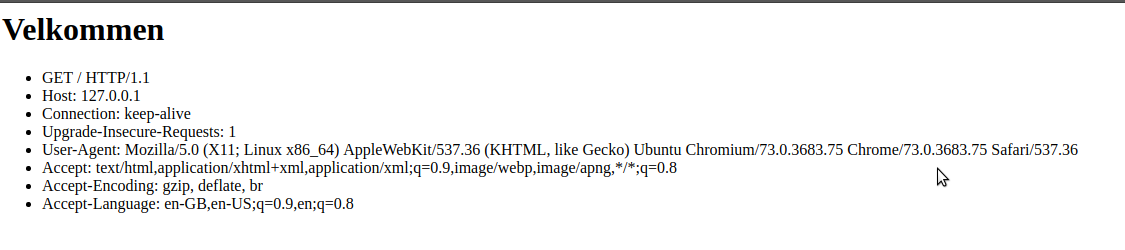
Kjører curl på serveren og får da



Altså skjer det en TCP handshake, en GET, en OK, og mutuell FIN.



Curl henter da nettsiden, men klager så på at den ikke får forbindelse, som er rart, men kan være pga. at serveren ikke bevarer forbindelse. Den laster derimot uten problem i chromium:



For chromium, ser pakkefangst ganske identisk ut, men det skjer en ny TCP oppkobling og en GET request for favicon, som da blir besvart med indekssiden på nytt da serveren ikke gjør noe sjekk på *hva* som bes om, og vil svare selv på en TCP pakke som bare er to newlines. (Uten noe innhold ville den da såklart ikke ha noen headers å fylle listen på siden med.)

## 2.5 Diskusjon

Den asynkrone kommunkasjonen gjør at flere kan koble seg opp samtidig, og TCP overleverer informasjon pålitelig. Det er såklart store pakkehoder, og overflødige ACK meldinger, men det er ganske lett å bruke og det er pålitelig.

# 3. Øving 4 – Kommunikasjon over UDP

## 3.1 Begreper

UDP – Transportprotokoll med mindre pakkehode enn TCP, som også mangler sjekk på om pakker kommer frem. Raskere, men mindre effektiv.

## 3.2 Programmeringsmetoder

Jeg har valgt å bruke C++ som programmeringsspråk, og biblioteket ASIO som er del av Boost (Boost::ASIO).

Generelt er fremgangsmåten basert på forrige øving, men en del detaljer blir forskjellig med UDP.

Namespacet boost::asio::ip::udp er mye brukt, og det er derfor “using boost::asio::ip::udp ” I toppen av filene. Da slipper jeg altså å skrive fullt namespace hver gang namespacet udp er brukt.

Det brukes også shared pointers, disse er I <memory> modulen I C++ sitt standardbibliotek. Disse er da pointers (referanser til hvor I minne noe er) med minnesikkerhet lagt oppå seg.

I denne øvingen velgte jeg å la klient kjøre helt synkront, da det oppstår en del detaljer ekstra i de asynkrone funksjonene, og klienten trenger ikke snakke med mer enn en socket.

Generelt brukes:  
boost::asio::io\_service  
udp::socket  
std::make\_shared<T>(CONSTRUCTOR ARGS)  
std::shared\_ptr<T>   
std::cout  
boost::asio::buffer  
boost::array<T, SIZE>   
socket.send\_to(buffer, endpoint, flag)  
socket.receive\_from(buffer, endpoint, flag)   
udp::endpoint  
std::string  
std::istream, std::ostream  
std::exception

For matteserver og webserver er det også:  
socket.async\_rececive\_from(buffer, endpoint, flag, callback)   
socket.async\_send\_to(buffer, endpoint, flag, callback)

Mens for matteklient:  
udp::resolver

## 3.3 Utdrag fra kode

### 3.3.1 Server

Likt forrige, så startes serveren ved å lage et objekt av WebServer og kjøre start() på det. Den konstruerer et endpoint på 8081, og lager en socket, og start starter så accept\_request() og io\_service.run().

Accept\_request ser da forskjellig ut. UDP mangler en acceptor som gjør 3-way-handshake, så det starter direkte med en melding. Jeg har da valgt å sende en enkelt byte fra klient for å begynne oppkoblingen. Man lager da et endpoint objekt, og ber socketen motta en melding. Om endpointet oppgitt ikke er initialisert enda, blir da melding fra hvem som helst å mottas, og endpointet vil bli satt til å være denne avsenderen. Tar da å gjør dette asynkront, og har callback metoden til å starte en ny asynkron accept\_request, og så lager den en ny socket, svarer klienten med den samme byten som ble mottatt så klienten kan få med seg den nye socketen som endpoint, og så starter den handle\_request loopen.

void accept\_request(){  
 auto remote\_endpoint = std::make\_shared<udp::endpoint>();  
 auto buf = std::make\_shared<boost::array<unsigned char, 1>>();  
  
 base\_socket.async\_receive\_from(boost::asio::buffer(\*buf), \*remote\_endpoint, 0, [=](boost::system::error\_code ec, std::size\_t s){  
 accept\_request();  
 if(!ec && (\*buf)[0] == 'h'){  
 std::cout << remote\_endpoint->address() << ':' << remote\_endpoint->port() << std::endl;  
 auto socket = std::make\_shared<udp::socket>(io\_service);  
 socket->open(udp::v4());  
 socket->send\_to(boost::asio::buffer(\*buf), \*remote\_endpoint, 0);  
 handle\_request(socket, remote\_endpoint);  
 }  
 });  
}

handle\_request tar da å mottar asynkront fra et spesifikt endepunkt. Callbacken rekurserer for å kunne motta neste melding fortløpende, tolker meldingen den mottar, og sender svar asynkront.

void handle\_request(std::shared\_ptr<udp::socket> const& socket, std::shared\_ptr<udp::endpoint> const& remote\_endpoint){  
 auto buf = std::make\_shared<boost::array<char, 256>>();  
 socket->async\_receive\_from(boost::asio::buffer(\*buf), \*remote\_endpoint, [=](const boost::system::error\_code &ec, size\_t s){  
 if(!ec){  
 std::string message\_in(buf->begin(), buf->begin()+s);  
 std::string failmessage;  
 if(message\_in == "exit") return;  
 else handle\_request(socket, remote\_endpoint);  
  
 [ Strengtolkning og så videre for kalkulator ]

std::string message\_out;  
  
 if(failmessage.length()){  
 message\_out = failmessage;  
 }  
 else {  
 message\_out = std::to\_string(isPlus ? a+b : a-b);  
 }  
 message\_out += "\r\n";  
 message\_out += '\0';  
  
 // Write to client  
 socket->async\_send\_to(boost::asio::buffer(message\_out), \*remote\_endpoint, [=](const boost::system::error\_code &ec, size\_t) {  
 // If not error:  
 if (!ec) {  
 std::cout << "server: reply sent to client" << std::endl;  
 std::cout << message\_out << std::endl;  
 }  
 });  
  
 }  
 });  
}

### 3.3.2 Klient

Konstruerer først et MathClient objekt med «127.0.0.1» og 8081 som adresse og port.

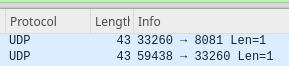
Konstruktør tar så å lager et resolver::query objekt for adresse og port, og får resolver til å konstruere et endpoint fra queriet. Så lages det en socket, og en boost::array<char, 1>, med bare ‘h’ i seg. Dette sendes da til endepunktet. Så lages ett nytt endepunkt, som brukes i receive\_from for å initialiseres med det nye endepunktet fra serveren for denne UDP forbindelsen. Den sjekker da at meldingen er ‘h’, og gir så socketen og det nye endepunktet til ask loopen.

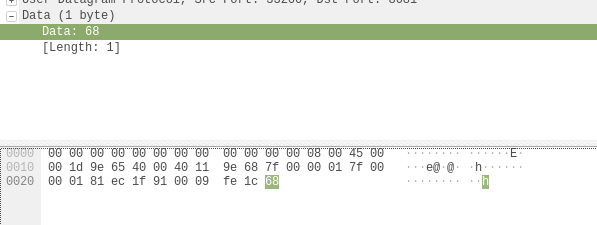
MathClient(const std::string &host, unsigned short port) : resolver(io\_service) {  
 // Create query from host and port  
 auto query = udp::resolver::query(host, std::to\_string(port));  
 udp::endpoint remote\_base(\*resolver.resolve(query));  
  
 udp::socket socket(io\_service);  
 socket.open(udp::v4());  
  
 boost::array<char, 1> buf0 = {'h'};  
 boost::array<char, 1> buf1;  
 boost::system::error\_code ec;  
  
 socket.send\_to(boost::asio::buffer(buf0), remote\_base);  
 udp::endpoint remote\_endpoint;  
 socket.receive\_from(boost::asio::buffer(buf1), remote\_endpoint);  
 if(buf1[0] == 'h'){  
 std::cout << remote\_endpoint.address() << ':' << remote\_endpoint.port() << std::endl;  
 std::string message\_out;  
 ask(socket, remote\_endpoint);  
 io\_service.run();  
 }  
}  
I ask loopen leses det inn tekst fra konsoll, teksten sendes til server, et svar hentes fra server, og svaret blir skrevet ut.

void ask(udp::socket &socket, udp::endpoint &remote\_endpoint){  
 std::string message\_out;  
 while( std::cin >> message\_out and message\_out != "exit"){  
 socket.send\_to(boost::asio::buffer(message\_out), remote\_endpoint);  
 boost::array<char, 256> message\_in;  
 socket.receive\_from(boost::asio::buffer(message\_in), remote\_endpoint);  
 std::cout << std::string(message\_in.data()) << std::endl;  
 ask(socket, remote\_endpoint);  
 }  
  
}

## 3.4 Pakkefangst

Starter opp server og klient. Ser da at de sender en UDP pakke hver vei. Den første sendes da fra en dynamisk port til 8081, og får svar fra en ny dynamisk port.

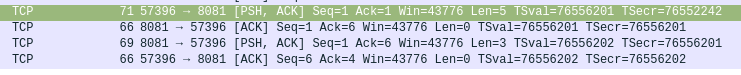


Dataen i begge er da ‘h’

Sender så «1+1» fra klient, og ser at det går en pakke hver vei, nå via de to dynamiske portene



Vi kan og merke at lengden på UDP pakkene er rundt 40-50 byte, med veldig kort data. Svaret er lengre, da det sendes tall som streng med et visst antall siffer, mens fra klient ble det sendt bare tre karakterer, ‘1’, ‘+’, ‘1’. Vi kan så sammenligne med pakkefangst av øving 3 sin matteklient, der den samme samtalen ender opp på 60-70 byte, eller 140 om man teller med ACK meldingene.



## 3.5 Diskusjon

UDP er da mindre, og sender halvparten så mange pakker. Det er derimot mulig for pakker å forsvinne uten at avsender er klar over det og kan sende pakken over på nytt, som kan forårsake kommunikasjonsfeil. For eksempel, I dette eksempelet kunne klienten ventet for evig på et svar, uten å gi bruker mulighet til å spørre på nytt, uten at server vet at noe er galt. Dette kan tas seg av i kode på forskjellige vis, f.eks. kunne jeg latt bruker spørre ett nytt spørsmål mens klient venter på svar, og da ville det nye spørsmålet kunne blitt svart på. Ellers kunne det blitt brukt timeout på svar, og spørsmålet kunne ha blitt sendt på nytt om det timer ut. Løsninger blir såklart på case-by-case basis, da den generelle løsningen er å bruke TCP istedenfor UDP, som ikke alltid er ideelt eller praktisk. Løsningene foreslått her avhenger da av at et svar forventes, og dette er da ofte ikke sant om man først har valgt UDP.

# 4. Øving 5 – Programmatisk bruk av databaser via ORM

## 4.1 Begreper

ORM – Object-Relational Mapping. Et begrep for når man har en strukturert kobling mellom objekter i kode (e.g. i C++ eller Java), og relasjoner i en relasjonsdatabase (e.g. MySQL eller SQLite). Eksempler på ORM verktøy er da ODB for C++, og Java Persistance i Java.

Reflection – et verktøy / en feature i visse programmeringsspråk, bl.a. Java og Python, der koden kan modifisere seg selv eller annen kode. Reflection mangler da i C og C++, da de er kompilert til maskinkode. Man kan derimot få til noen lignende ting ved å analysere og generere kode før kompilering.

## 4.2 Programmeringsmetoder

Jeg har valgt å bruke C++ som programmeringsspråk, og ODB som verktøy for ORM.

### 4.2.1 Kompilering

Siden C++ mangler Reflection, da det er et fullstendig kompilert språk, er det veldig vanskelig å få til et typisk ORM rammeverk i det. ODB har da valgt å bruke preprocessor og lignende til det arbeidet som reflection gjør. Kode må derfor kjøres gjennom ODB sin egen C++ til C++ compiler for å lage databaseobjektene, som kompliserer kompilasjon litt. For å få dette til i CMAKE brukte jeg da <https://github.com/BtbN/OdbCmake>. Stegene jeg måtte ta:

Legge filen database.h fra repoet til prosjektet.  
Basere CMakeLists på den i repoet  
 Legge mappen cmake/Modules fra repoet til i prosjektet, som da har filene FindODB.cmake og UseODB.cmake, som forteller CMake hvordan den skal finne og bruke pakken ODB  
Inkludere lisensfilen fra repoet som Odbmake\_LICENSE.txt, da den eneste begrensningen i lisensen er at lisensen må inkluderes

### 4.2.2 ORM objekter i ODB

ORM objekter defineres da i odb ved å sette #pragma db flagg over klasser og felter som skal kobles mot database, og legge til «friend class odb::access», som lar odb få tilgang til private felter i klassen.

Dette inkluderer da å spesifisere at en klasse er et db objekt, og om klassen bruker optimisitisk låsing.

Du kan og spesifisere informasjon om feltene, slik som at noe er id, at id skal auto\_incrementes, default verdier, nøyaktig type en variabel skal ha i databasen (f.eks. varchar(X)/text på strenger)

Når ODB da preprosesserer vil den generere støttekoden rundt denne klassen, og en sql fil man kan bruke for å skape tabellen i databasen.

Man må da inkludere den genererte headerfilen i tillegg til selve klassen sin header når det skal brukes. Man kan få en klage i IDE på at den ikke finnes, men når man har kjørt den en gang tar ihvertfall CLion å innser at filen finnes i cmake-build-debug/odb\_gen. Man må så klart også inkludere <odb/database.hxx> og <odb/mysql/database.hxx> for bindingene. Erstatt evt. Mysql med et annet støttet databasespråk.

### 4.2.3 Objekter/Klasser/Metoder brukt:

Std::cin, std::cout  
std::string  
odb::mysql::database  
odb::transaction  
#pragma db

## 4.3 Utdrag fra kode

### 4.3.1 Definisjon av Account tabellen i C++

Her brukes da #pragma db som markør til preprossessor om at dette er ODB flagg. Object brukes da til å si det er et ORM objekt. Optimistic betyr at optimistisk låsing skal brukes. Dette må da fjernes for oppgave 3 der man skal få feil, og må inkluderes for oppgave 4 der optimistisk låsing skal la oss oppdage og angre feilen. Som nevnt er friend class odb::access for å la odb få endre på interne variabler, da odb ikke bruker getters og setters via reflections slik som i javabeans. Det kreves også av odb at det finnes en konstruktør uten argumenter, men denne kan da være privat.

#pragma db object optimistic  
class account{  
private:  
 friend class odb::access;  
 account(){};

Det brukes så “id” til å markere at dette er id for tabellen, og «auto» for auto increment. Default(0) setter defaultverdi, type(Str) setter typen spesifikt istedenfor implisitt. Version brukes til å si at dette er feltet brukt for optimistic locking.

#pragma db id auto  
 int \_account\_number;  
  
 #pragma db default(0)  
 double \_balance;  
 #pragma db type("VARCHAR(100)")  
 string \_name;  
  
 #pragma db version default(0)  
 unsigned' long version;

Det er da ingen krav om getters og setters i odb, men har her brukt det som virker som et vanlig c++ format, der du bruker variabelnavnet som funksjon, og legger en understrek foran det ekte navnet. ODB virker til å gjenkjenne dette formatet, da understrekene foran ikke tas med i den genererte sql filen.

public: // getters og setters:  
 const int &account\_number() const{ return \_account\_number; }  
 const double &balance() const{return \_balance; }  
 void balance(const double &v){ \_balance=v; }  
 const string &name() const{ return \_name; }  
 void name(const string &v) { \_name=v; }  
 account(const string &name) : \_name(name){}; //konstruktør  
};

Det genereres som nevnt en tilsvarende sql fil, account.sql:

/\* This file was generated by ODB, object-relational mapping (ORM)  
 \* compiler for C++.  
 \*/  
  
DROP TABLE IF EXISTS `account`;  
  
CREATE TABLE `account` (  
 `account\_number` INT NOT NULL PRIMARY KEY AUTO\_INCREMENT,  
 `balance` DOUBLE NOT NULL DEFAULT 0,  
 `name` VARCHAR(100) NOT NULL,  
 `version` BIGINT UNSIGNED NOT NULL DEFAULT 0)  
 ENGINE=InnoDB;

### 4.3.2 Oppgave 2 – skapelse, lasting og persistence

Skaper først databaseobjektet, forteller den at den skal bruke bruker test, med tomt passord, på databasen «datacom5». Man kan videre evt. spesifisere en port, socket, charset, og noen flagg.

odb::mysql::database db("bruker", "passord", "database", "url");

Så lager jeg en blokk, bare for å ha et scope for transaksjonene å falle ut av. Jeg lager instanser av objektene spesifisert i account.hpp, starter en transaksjon, og ber databasen persiste objektene. Til slutt committer jeg.

{  
 account john("john");  
 john.balance(50.2);  
 account jane("jane");  
  
 odb::transaction t(db.begin());  
  
 db.persist(john);  
 db.persist(jane);  
  
 t.commit();  
}

Så får jeg applikasjonen til å vente så jeg kan sjekke endringer i mysql

string waiter;  
cin >> waiter;

Så starter jeg en ny transaksjon, henter alle kontoer der balansen er større enn 25, setter de til 25.3, og committer. Merk at odb har en veldig fin måte å querye på, som bruker prepared statements i bakenden. Antakeligvis har odb::query<account> et statisk objekt for hvert felt, med en override på > operatoren som returnerer en comparator funksjon eller lignende.

{  
 odb::transaction t(db.begin());  
  
 for(account &e : db.query<account>(odb::query<account>::balance > 25)){  
 e.balance(25.3);  
 db.update(e);  
 }  
 t.commit();  
}

### 4.3.3 Oppgave 3 og 4 – Race condition skrivefeil, og optimistisk låsing

Henter første bruker med navn john, og første bruker med navn jane.

odb::transaction t(db.begin());  
account john = \*db.query\_one<account>(odb::query<account>::name == "john");  
account jane = \*db.query\_one<account>(odb::query<account>::name == "jane");  
t.commit();

Venter så på input for å kunne tvinge fram race conditions til å skje den vei jeg ønsker.  
Så endrer jeg verdien til john i en commit. Merk at man ikke kan skru på autocommit i odb, så objekter oppdateres ikke i database uten en commit.

odb::transaction t2(db.begin());  
cout << "changing john" << endl;  
john.balance(john.balance() + 413.612);  
db.update(john);  
t2.commit();

Venter så igjen, for så å oppdatere jane. Jane oppdateres derimot I en try/catch, for å fasilitere øving 4. For øving 3 er da bare try blokken relevant, der jane oppdateres på lignende vis som john.

try {  
 odb::transaction t2(db.begin());  
  
 cout << "changing jane";  
 jane.balance(jane.balance() - 413.612);  
 db.update(jane);  
  
 t2.commit();  
}

I øving 4 vil da dette feile og kaste en odb::object\_changed exception dersom det har skjedd en versjonsfeil, da optimistic flagget er på klassen. Om denne feilen skjer, tar vi da å fanger den, og angrer endringer på john. Vi tar så å kaster feilen igjen, da vi fortsatt har hatt en feil under kjøring som burde bli informert om. Denne kunne eventuelt vært fanget opp ett nivå opp, og ført til at funksjonen blir kalt på nytt.

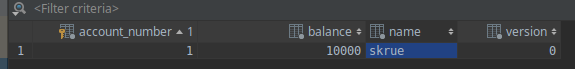
catch (odb::object\_changed &e) {  
 odb::transaction t2(db.begin());  
 odb::result<account> r3 = db.query<account>(odb::query<account>::account\_number == john.account\_number());  
 account john2 = \*r3.begin();  
 john2.balance(john2.balance() - 413.612);  
 db.update(john2);  
 t2.commit();  
 throw e;  
}

## 4.4 Pakkefangst og MySQL verdier under kjøring

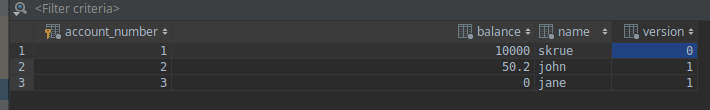
### 4.4.1 Oppgave 2

Har valgt å bruke mysql databasen til skolen, da jeg ikke fant noe på pakkefangst mot lokal database

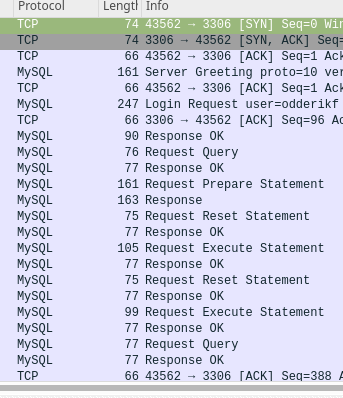
Starter med en tom tabell, pluss én konto.



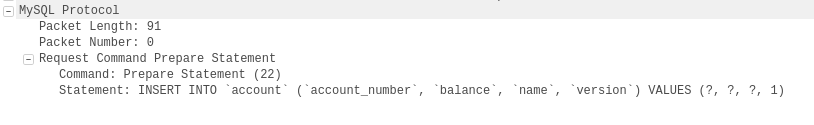
Kjører så første del av oppgave 2, der objektene lages og så persistes. Vi ser da at john og jane har blitt opprettet i databasen:

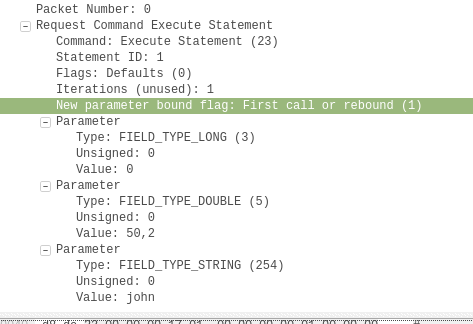


På pakkefangst ser vi da at det gjøres en TCP handshake, og så brukes applikasjonslagprotokollen MySQL. Vi ser at den preparer en statement, og så gjentatte ganger resetter den så verdiene er ufylt, og så sender med verdiene i en Execute Statement pakke.

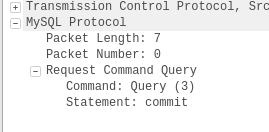


Vi ser da på den første Prepare Statement forespørselen, og ser at den er for INSERT setningen for john og jane.

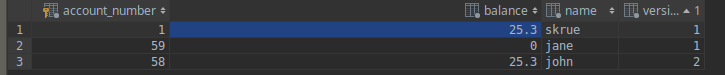
  
Første execute holder da verdier for john:



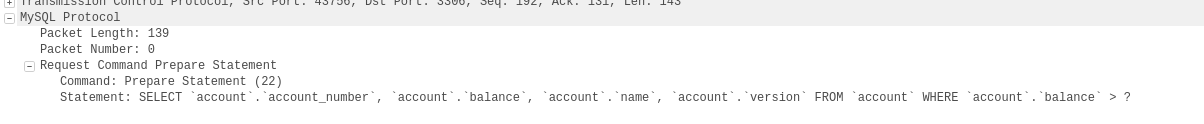
Og den andre holder på liknende vis verdier for jane. Request query virker da til å være begin og commit statementene før og etter.



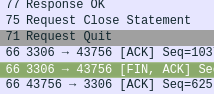
Kjører så resten av oppgave 2, der vi oppdaterer balansen til de med over 25 kroner, og ser da at skrue også har fått sin konto oppdatert.



Vi ser på pakkefangst mye av det samme. En ny Request Query for begin, en Prepare statement for select setningen:

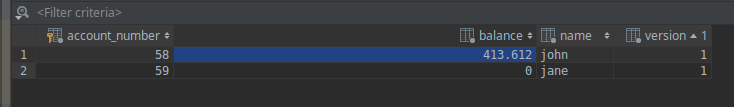


Og så reset, execute, der execute sender over verdien 25. Så Prepares en UPDATE setning, som så resettes og kjøres to ganger, en for john og en for skrue. Til slutt kommer en Request Query med commiten, og så en Request Close Statement og en Request Quit, som blir svart av serveren med FIN. Antakeligvis fordi databasen gikk ut av scope.

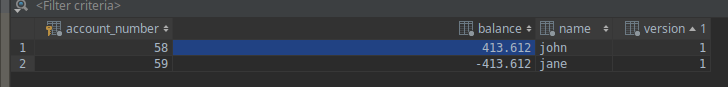


### 4.4.2 Oppgave 3

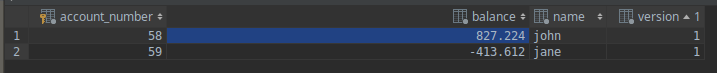
Kjører ikke pakkefangst her, da det blir mye av det samme, og MySQL protokollen ikke er pensum. Setter først saldo på jane og john til 0. Starter oppgave 3, og lar den fortsette en gang, så den har oppdatert john. Vi ser at john har mottatt penger



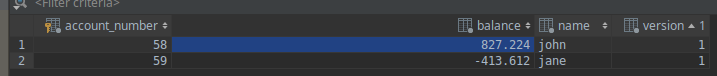
Vi starter så en ny instans av oppgave 3, så den leser verdier midt i forrige kjøring. Vi lar så den første kjøringen bli ferdig, og ser at john og jane har rette verdier



Vi lar så andre kjøring endre john, som gir han en ny verdi som den skal



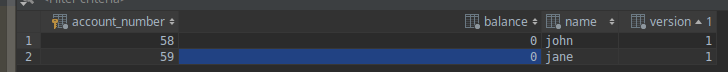
Men når vi så endrer jane, endrer vi henne med tanke på at hun hadde 0kr. Men siden vi sist leste hennes saldo, har hun gått ned 413.612kr, og skulle altså vært på -827.224. Her ser vi da at 413.612 kroner i gjeld har forsvunnet.



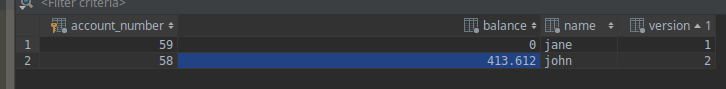
### 4.4.3 Oppgave 4

Legger så på optimistic og version flaggene igjen i account.hpp og kjører oppgave3 igjen, på nøyaktig samme vis.

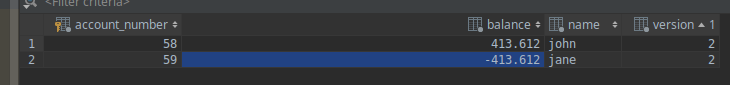
Før endring:



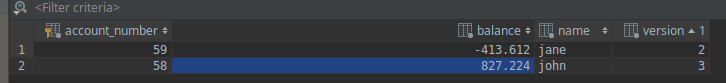
Etter første kjøring endrer john.



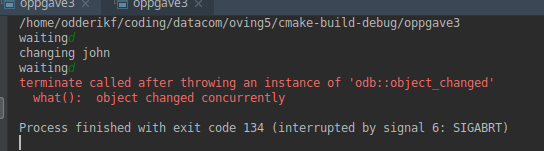
Starter andre kjøring, lar den lese, og så lar første kjøring fullføre. Merk at versjonsnummeret har endret seg



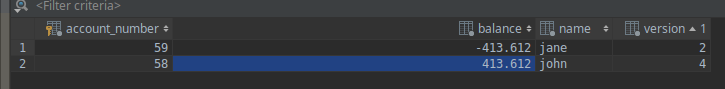
Lar andre kjøring endre john og ser at vi får rimelig saldo på john



Lar den fullføre og ser vi får kastet en feil



Og at endringene fra andre kjøring er blitt reversert:



## 4.5 Diskusjon

ORM er da en ganske grei måte å oversette databaser til objekter og tilbake på. C++ bindingene er også ganske grei å jobbe med, dog litt rar å kompilere. ODB virker til å ha god nok syntaks til å spesifisere nøyaktige typer, felter, etc til å kunne retrofitte det til eksisterende database, men det tar også å genereres definisjoner for tabeller om man ønsker å ha kodeklassene som definitiv versjon istedenfor å lage database først. Den virker til å være grundig med bruk av Prepared Statements, og er derfor sikret mot SQL Injection.

# 5. Øving 6 – Websocket

## 5.1 Begreper

Websocket – en protokoll på applikasjonslaget for toveis-kommunikasjon. Starter som en HTTP GET forespørsel med noen header flagg som ber om å oppgradere, og serveren svarer da med relevante felter om dette er støttet, og deretter kan man snakke direkte.

## 5.2 Programmeringsmetoder

Jeg har valgt å bruke C++ som programmeringsspråk, og biblioteket ASIO som er del av Boost (Boost::ASIO). Jeg har i tillegg brukt openssl for sin sha funksjon. For kompilering med OpenSSL kan man da i CMake linke alt med:

find\_package(OpenSSL REQUIRED)  
IF(OPENSSL\_FOUND)  
 SET(OPENSSL\_USE\_STATIC\_LIBS true)  
ENDIF()  
target\_link\_libraries(oving6 OpenSSL::Crypto)

Jeg bruker da boost::asio::ip::tcp som tidligere, og har «using boost::asio::ip::tcp» på toppen av filen.

Serveren sin struktur er da lignende den i øving 3.

Klasser/objekter/funksjoner brukt:  
std::vector  
std::shared\_ptr  
std::make\_shared<T>(CONSTRUCTOR ARGS)  
std::ostream  
std::map  
std::getline   
boost::algorithm::trim(string)  
SHA1(key, keysize, digest)

boost::asio::io\_service   
tcp::endpoint  
tcp::acceptor  
tcp::socket  
boost::asio::async\_read\_until(socket, buffer, end\_str, callback)  
boost::asio::async\_write(socket, buffer, callback)  
boost::asio::streambuf

## 5.3 Utdrag fra kode

Jeg tar som tidligere først å lager et WebServer objekt, og kjører start på det. Konstruktør lager et endpoint og en acceptor. Start kjører accept\_request og så io\_service. Det er da også noen magiske tall involvert i oppgaven. MAGIC\_GUID er da den oppgitt i RFC standarden for websockets, mens FIN, OP\_TEXTFRAME og OP\_CLOSE er bitflagg for første byte i frames på websocket.

#define MAGIC\_GUID "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"  
#define FIN 0b10000000  
#define OP\_TEXTFRAME 0b0001  
#define OP\_CLOSE 0b1000

Accept request lager en socket, lytter på den, og i callbacken gir den så socketen videre til handle\_request

void accept\_request(){  
 auto socket = std::make\_shared<tcp::socket>(io\_service);  
 acceptor.async\_accept(\*socket, [this, socket](const boost::system::error\_code &ec){  
 accept\_request();  
 if(!ec){  
 handle\_request(socket);  
 }  
 });  
}

Handle request leser så hele http requesten asynkront. Den tar så å sjekker at det er en GET /hello request, og så leser den inn alle headers til et map (dictionary / hashmap)

auto read\_buffer = std::make\_shared<boost::asio::streambuf>();  
boost::asio::async\_read\_until(\*socket, \*read\_buffer, "\r\n\r\n", [=](const boost::system::error\_code &ec, size\_t){  
 if(!ec){  
 std::string message;  
 std::string failmessage;  
 std::istream read\_stream(read\_buffer.get());  
 message.pop\_back();  
 if(message == "exit") return;  
 read\_stream >> message;  
 if(message != "GET") return;  
 read\_stream >> message;  
 if(message != "/hello") return;  
 std::getline(read\_stream, message); // get rid of rest of line  
  
 std::map<std::string, std::string> headers;  
 while(std::getline(read\_stream, message) && message != "\r"){  
 [ splitter på : og trimmer vekk whitespace ]   
 headers[title] = value;  
 }

Så sjekker den at alle headerne finnes og deres verdier stemmer med RFC standarden for websockets

if(headers.find("Host") == headers.end()) return;  
if(headers.find("Upgrade") == headers.end()) return;  
if(headers["Upgrade"] != "websocket") return;  
if(headers.find("Connection") == headers.end()) return;  
if(headers["Connection"] != "Upgrade") return;  
if(headers.find("Sec-WebSocket-Key") == headers.end()) return;  
if(headers.find("Sec-WebSocket-Version") == headers.end()) return;  
if(headers["Sec-WebSocket-Version"] != "13") return;

Så genererer den digesten til websocket key-en:

std::string sec\_websocket\_key = headers["Sec-WebSocket-Key"];  
sec\_websocket\_key += MAGIC\_GUID;   
unsigned char digest[SHA\_DIGEST\_LENGTH] = {0};  
SHA1((unsigned char \*)sec\_websocket\_key.data(), sec\_websocket\_key.size(), (unsigned char \*)&digest);

Og base-64 enkoder den.

std::string accept\_key;  
  
char alphabet[64] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',  
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',  
 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',  
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',  
 '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'};  
  
int group24bit[7];  
for(int i = 0; i < 6; ++i){  
 group24bit[i] = digest[3\*i];  
 group24bit[i] <<= 8;  
 group24bit[i] |= digest[3\*i+1];  
 group24bit[i] <<= 8;  
 group24bit[i] |= digest[3\*i+2];  
}  
group24bit[6] = digest[3\*6];  
group24bit[6] <<= 8;  
group24bit[6] |= digest[3\*6+1];  
group24bit[6] <<= 8;  
for(int i = 0; i < 6; ++i){  
 accept\_key += alphabet[group24bit[i] >> 18];  
 accept\_key += alphabet[group24bit[i] >> 12 & 0b00111111];  
 accept\_key += alphabet[group24bit[i] >> 6 & 0b00111111];  
 accept\_key += alphabet[group24bit[i] & 0b00111111];  
}  
  
accept\_key += alphabet[group24bit[6] >> 18];  
accept\_key += alphabet[group24bit[6] >> 12 & 0b00111111];  
accept\_key += alphabet[group24bit[6] >> 6 & 0b00111111];  
accept\_key += '='; //special case instead of the 6 bits that are padded zeroes

Så kan vi endelig lage handshaken

accept\_key += "\r\n";  
std::string handshake =  
 "HTTP/1.1 101 Switching Protocols\r\n"  
 "Upgrade: websocket\r\n"  
 "Connection: Upgrade\r\n"  
 "Sec-WebSocket-Accept: ";  
handshake += accept\_key;  
handshake += "\r\n";

Og sende den til klienten

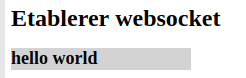
auto write\_buffer = std::make\_shared<boost::asio::streambuf>();  
std::ostream write\_stream(write\_buffer.get());  
write\_stream << handshake;  
  
async\_write(\*socket, \*write\_buffer, [this, socket, write\_buffer](const boost::system::error\_code &ec, size\_t) {  
 if (!ec) {  
 std::cout << "server: Switching Protocols with client" << std::endl;  
 std::string temp;  
 handle\_connection(socket);  
 }  
});

I handle connection tar vi da å sender første melding over websocketen, og så etterlater vi videre kommunikasjon til listen\_connection. Listen\_connection er da ikke ferdig skrevet da den delen var valgfri, oppgaven ba kun om å kunne sende denne første meldingen fra serveren. Listen\_connection vil derfor heller ikke dokumenterers her.

void handle\_connection(const std::shared\_ptr<tcp::socket> &socket){  
 auto write\_buffer = std::make\_shared<boost::asio::streambuf>();  
 std::ostream write\_stream(write\_buffer.get());  
 std::string message = "hello world";  
 write\_stream << (unsigned char) (FIN | OP\_TEXTFRAME);  
 write\_stream << (unsigned char) message.size();  
 write\_stream << message;  
 boost::asio::async\_write(\*socket, \*write\_buffer, [=](const boost::system::error\_code &ec, size\_t) {  
 if (!ec) {  
 std::cout << "server: sent hello world to client" << std::endl;  
 listen\_connection(socket);  
 }  
 });  
}

For å oppsummere tar da listen\_connection og leser inn 2 byte fra socketen, ser hva slags op kode dette er, stenger oppkoblingen om det er OP\_CLOSE, ellers sjekker lengden og leser videre det den må for å motta resten av meldingen. Den ville så måtte demaskert meldingen, som er der jeg sluttet å jobbe på den. Til slutt tar den da og forsøker sende meldingen videre til alle oppkoblede websockets, inkudert avsender.

Lagde så et test-script i form av en HTML fil med script tag. Den ser da slik ut:

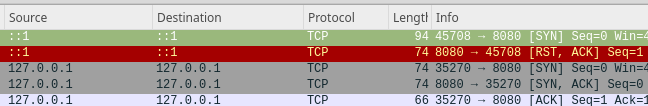


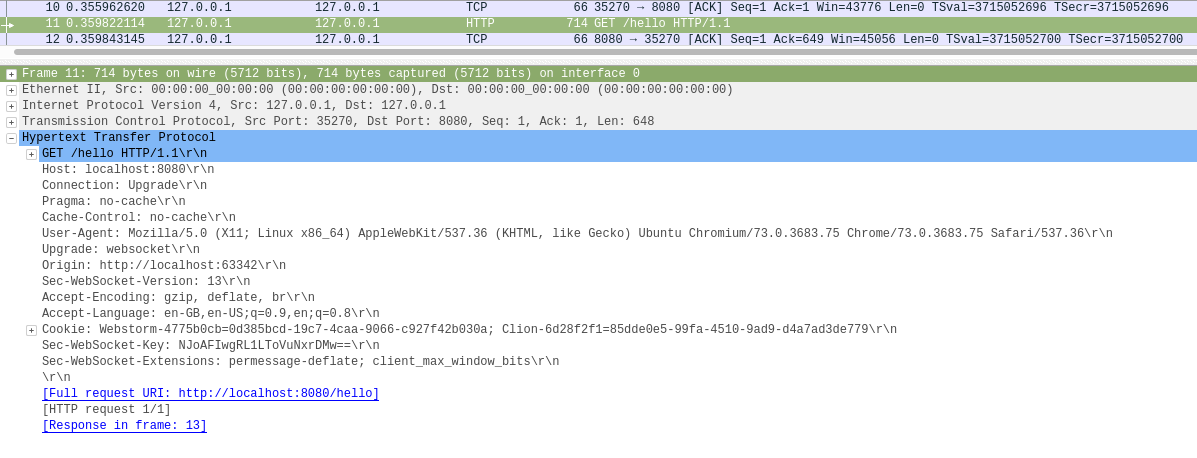
Den gråe boksen starter tom, men blir endret til innholdet i pakken hver gang en websocket pakke sendes. Alternativt settes den til Failed om noe går galt.

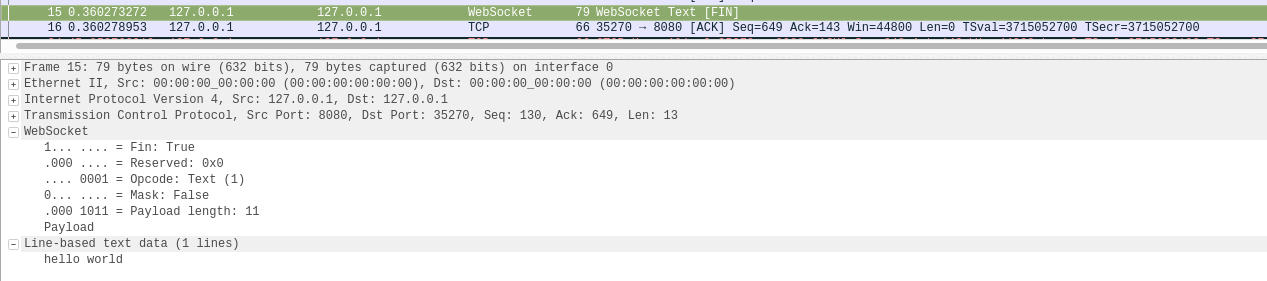
<!DOCTYPE html>  
<html lang="en">  
<head>  
 <meta charset="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
 <h1>Etablerer websocket</h1>  
 <h2 style="background-color: lightgrey;width: 10em;height: 1.2em;" id="responsebox"></h2>  
 <script>  
 let ws = new WebSocket('ws://localhost:8080/hello');  
 ws.onopen = e => console.log("open");  
 ws.onmessage = e => {  
 console.log(e.data);  
 document.getElementById("responsebox").innerText = e.data;  
 };  
 ws.onerror = e => {  
 document.getElementById("responsebox").innerText = "Failed";  
 }  
 </script>  
</body>  
</html>

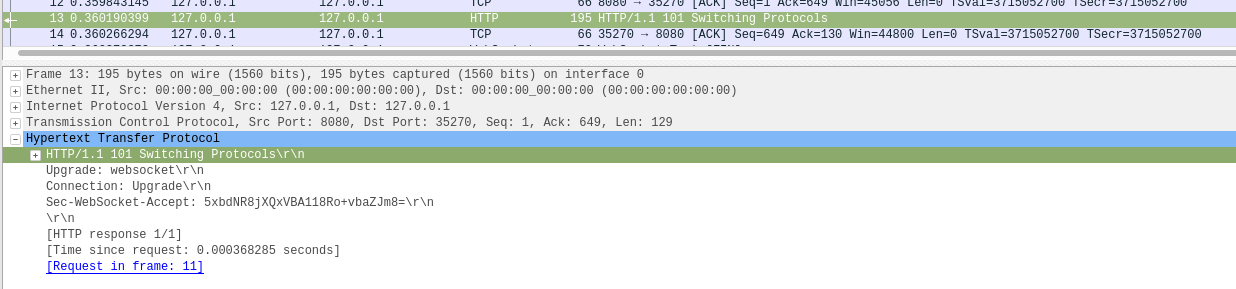
## 5.4 Pakkefangst

Starter server. Laster så HTML filen i chromium. Merk at filen ikke lastes via programmet, men via CLion på port 63342, så første request vi ser her er da ikke fra nettleser som laster filen, men fra javascript sin websocket som kobler seg mot localhost 8080. Ser at vi først får en SYN, RST/ACK over ipv6. Det kjøres så suksessfull three-way handshake over ipv4.

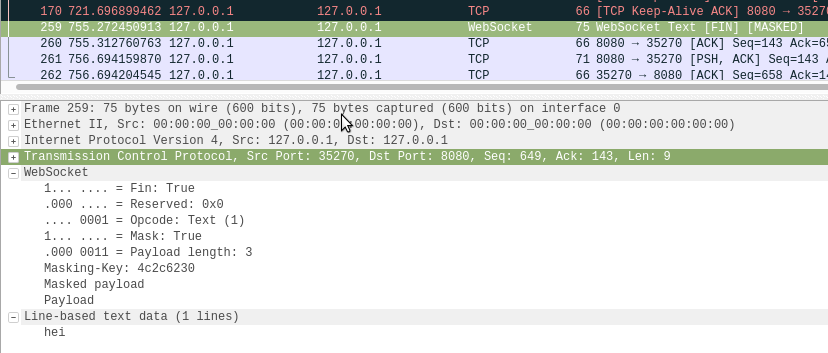


Deretter kjøres det en GET /hello http/1.1

Vi ser da at den har WebSocket sine headerfelter, bl.a. Upgrade: websocket\r\n. Det svares så med en Switching Protocols fra serveren, med en sec-websocket accept og andre felter

Til slutt kommer det her en websocket text, med FIN flagg da hele sendes i én melding. Vi ser at maske er false, payload lengden er oppgitt, opcode er text, og fin er true. Meldingen har da blitt sendt i klartekst.

Videre sendes det en drøss Keep-Alive meldinger fra JS om å sjekke status. Når jeg da kjører «ws.send(‘hei’) i js konsoll på siden, ser jeg da at det sendes med MASKED flagget. Wireshark demaskerer da meldingen for oss, og vi kan se at den er hei. Serveren svarer da ikke meningsfullt, da jeg som nevnt ikke har gjort ferdig den valgfrie delen av øvingen.



## 5.5 Diskusjon

Jeg ser da at WebSocket er en grei måte å sende ganske småe meldinger frem og tilbake toveis på. Det er derimot veldig tungt å skrive bibliotek for det, da det er veldig mange småting som må til for bl.a. protokollswitching og demaskering, og dessuten er det å lese en del av en frame og så tolke bits litt verbost i C++, og tok en stund å få til rett.