

# IDATT2104 - Datakom – Arbeidskrav 2

## Innholdsfortegnelse

Øving 3 .....	2
Oppgaven .....	2
Programkode .....	2
Lagene: .....	5
Klient-tjener .....	5
Three way handshake: .....	5
Dataoverføring .....	6
Tilkoblingen avsluttes .....	6
webserver .....	6
Øving 4 .....	9
Oppgaven .....	9
UDP kalkulator .....	9
programkode .....	9
UDP kommunikasjon .....	13
TLS .....	13
Programkode .....	13
Lagene: .....	16
Kommunikasjon .....	16
Nedkobling .....	16
Tjenerens sertifikat .....	17
Kryptografisuite .....	17
Hvordan sesjonsnøkler opprettes .....	18

Oppgaven ble utført alene og alt er derfor gjort i localhost. Dette gjør at alle ip-adresser samt at det ikke er tatt i bruk noen mac adresser for overføring av data.

Min lokale IPv4 adresse er: 127.0.0.1

Min mac adresse er: ether f0:18:98:2b:4d:ce

## Øving 3

### Oppgaven

Oppgaven gikk ut på å lage en nettverksapplikasjon basert på klient/tjener-modellen. I del 1 skulle det utvikles en enkel klient/tjener-applikasjon hvor klienten sender to tall til tjeneren, hvor tjener da skal enten addere eller subtrahere tallene basert på hva bruker ønsker.

Tjeneren utfører deretter den ønskede operasjonen og returnerer resultatet tilbake til klienten. Tjeneren skal også sende tilbake passende feilmelding om feil skulle oppstå.

Interaksjonen skal kunne gjentas slik at flere beregninger kan utføres i løkke.

Videre skal tjenerapplikasjonen utvides til å kunne håndtere flere samtidige klientforbindelser. Dette oppnås ved å implementere multitråding på tjenersiden, hvor hver klientforbindelse betjenes av en egen tråd.

Del 2 av Oppgaven innebærer å lage en enkel webtjener som håndterer én klient om gangen. Når en klient kobler seg til, skal tjeneren returnere en HTML-side med en `<H1>` velkomstmelding og en punktliste (`<UL>` med `<LI>` elementer) som viser klientens HTTP-header.

### Programkode

Applikasjonen er en enkel nettverksapplikasjon laget i C++. Den består av de tre komponentene: Client, Server og Webserver.

Klientkoden din oppretter en TCP-forbindelse til serveren og sender deretter data basert på brukerinput og mottar svar.

**Opprette Socket:** Koden starter med å opprette en socket med «AF\_INET» for IPv4 og TCP-kommunikasjon. Hvis socket ikke kan opprettes, vises en feilmelding, og programmet avslutter.

**Konfigurere Serveradressen:** Den setter opp serverens adresseinformasjon, inkludert IP-adressen (her er det "127.0.0.1" for localhost) og portnummeret (8080). Denne adressen forteller klienten hvor den skal koble til.

**Koble til Serveren:** connect() forsøker å koble til serveren ved hjelp av den tidligere konfigurerte adressen. Ved en feil i tilkoblingsforsøket returneres et negativt tall, som avslutter programmet med en feilmelding.

**Kommunikasjon med Serveren:** Klienten går inn i en løkke hvor den venter på brukerinput. Brukeren kan enten skrive en operasjon eller 'q' for å avslutte. Inputen sendes til serveren, og klienten venter deretter på et svar. Når svaret mottas, vises det for brukeren. Bufferen renses etter hver melding for å forhindre datalekkasje mellom meldingene.

**Avslutte Forbindelsen:** Når brukeren bestemmer seg for å avslutte ved å skrive «q», lukker klienten socketen og avslutter programmet.

```
int main() {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char buffer[1024] = {0};

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cout << "\n Socket creation error \n";
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        std::cout << "\nInvalid address/ Address not supported \n";
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cout << "\nConnection Failed \n";
        return -1;
    }

    while (true) {
        std::string input;
        std::cout << "Enter operation or 'q' to exit: ";
        std::getline(& std::cin, & input);

        if (input == "q") {
            break;
        }

        send(sock, input.c_str(), strlen(& input.c_str()), 0);
        valread = read(sock, buffer, 1024);
        std::cout << "Result: " << buffer << std::endl;
        memset(& buffer, 0, len: sizeof(buffer)); // Rens bufferen for neste melding
    }

    close(sock);
    return 0;
}
```

Figur 1: Client

Serveren flertrådet TCP-server som lytter på port 8080, aksepterer innkommende forbindelser, og håndterer hver klientforbindelse i sin egen tråd.

**Opprette og Konfigurere Socket:** Serveren oppretter en socket og konfigurerer den for å tillate gjenbruk av adresse/port for å unngå "address already in use" feil. Dette gjøres med «socket()» og «setsockopt()».

**Binde Socket til en Port:** Serveren binder den opprettede socketen til porten slik at den kan lytte etter innkommende forbindelser på denne porten. «bind()» funksjonen brukes til dette formålet.

**Lytt etter Innkommende Forbindelser:** Med «listen()», begynner serveren å lytte på den angitte porten, klar til å akseptere innkommende forbindelser.

**Akseptere Forbindelser og Opprette Tråder:** I en uendelig løkke, venter serveren på nye klientforbindelser med «accept()». For hver ny forbindelse som aksepteres, opprettes en ny tråd ved hjelp av «pthread\_create()». Denne tråden vil håndtere kommunikasjonen med den tilkoblede klienten.

**Håndtering av Klientforbindelser:** «handleClient» funksjonen kjøres i en separat tråd for hver klient. Den leser klientens forespørsel, tolker den, utfører en beregning basert på forespørselen og sender resultatet tilbake til klienten. Hvis klienten sender "q" eller lukker forbindelsen, avsluttes tråden.

```
1  #include <stdio.h>
2
3  #define PORT 8888
4
5  void* handleClient(void* socket_desc) {
6      int sock = *(int*)socket_desc;
7      free(socket_desc);
8      char buffer[1024] = {0};
9      char result[1024];
10     int valread;
11
12     while(true) {
13         memset(&buffer, 0, sizeof(buffer));
14         valread = read(sock, buffer, sizeof(buffer));
15         if (strcmp(buffer, "q") == 0 || valread == 0) {
16             printf("Forbindelse avsluttet av klienten.\n");
17             break;
18         }
19
20         int tall1, tall2;
21         char operasjon;
22         sscanf(buffer, "%d %c %d", &tall1, &operasjon, &tall2);
23
24         int resultat = (operasjon == '+') ? (tall1 + tall2) : (tall1 - tall2);
25         sprintf(result, "%d", resultat);
26         send(sock, result, strlen(result), 0);
27     }
28 }
29
30 close(sock);
31 return nullptr;
32 }
33
34 int main() {
35     int server_fd, new_socket;
36     struct sockaddr_in address;
37     int opt = 1;
38     int addrlen = sizeof(address);
39
40     server_fd = socket(AF_INET, SOCK_STREAM, 0);
41     setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt));
42     address.sin_family = AF_INET;
43     address.sin_addr.s_addr = INADDR_ANY;
44     address.sin_port = htons(PORT);
45
46     bind(server_fd, (struct sockaddr *)&address, sizeof(address));
47     listen(server_fd, 3);
48
49     while(true) {
50         printf("Venter på ny forbindelse...\n");
51         new_socket = accept(server_fd, (struct sockaddr *)&address, (&socklen_t*)&addrlen);
52         if (new_socket < 0) {
53             perror("accept");
54             continue;
55         }
56         printf("Forbindelse etablert.\n");
57
58         int* new_sock = new int(new_socket);
59
60         pthread_t thread;
61         if (pthread_create(&thread, nullptr, handleClient, (void*)new_sock) != 0) {
62             perror("could not create thread");
63         }
64         pthread_detach(thread); // Detach tråden slik at ressurser frigjøres når den avsluttes
65     }
66
67     close(server_fd);
68     return 0;
69 }
```

Figur 2: Server

Lagene:

## Transportlaget

I koden brukes TCP. TCP, som står for Transmission Control Protocol, er en nøkkelkomponent i internettets transportlag og spiller en viktig rolle i å sikre pålitelig, tilkoblingsorientert kommunikasjon mellom enheter på nettet. Ved å bruke TCP, kan applikasjoner som webservere og klienter utveksle data med garantier om at informasjonen ankommer i riktig rekkefølge, uten feil, og på en kontrollert måte som forhindrer overbelastning av nettverket. Ved å se på bruken av «SOCK\_STREAM» kan en se at det brukes TCP.

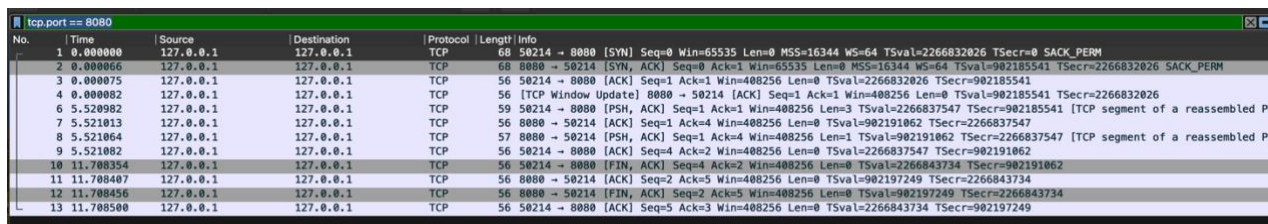
## Nettverkslaget

Koden bruker også internett protokollen. På nettverkslaget, fungerer Internet Protocol, som ryggraden i dataoverføring over internett. IP sørger for adressering og ruting av datapakker fra sender til mottaker over diverse nettverk. Det gjør det mulig for pakker å navigere gjennom komplekse nettverk ved å bruke IP-adresser, som unikt identifiserer hver enhet på nettet. Selv om IP ikke garanterer pålitelig levering på egen hånd, kombineres det effektivt med TCP for å gi en robust og effektiv ende-til-ende forbindelse.

En kan se at det brukes ipv4 i koden ved å se på «AF\_INET»

## Klient-tjener

Bildet viser kommunikasjon mellom klient og tjener



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	68	50214 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=2266832026 TSecr=0 SACK_PERM
2	0.000066	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=902185541 TSecr=2266832026 SACK_PERM
3	0.000075	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=1 Ack=1 Win=488256 Len=0 TSval=2266832026 TSecr=902185541
4	0.000082	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8080 → 50214 [ACK] Seq=1 Ack=1 Win=488256 Len=0 TSval=902185541 TSecr=2266832026
6	5.520982	127.0.0.1	127.0.0.1	TCP	59	50214 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=488256 Len=3 TSval=2266837547 TSecr=902185541 [TCP segment of a reassembled PDU]
7	5.521013	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [ACK] Seq=1 Ack=4 Win=488256 Len=0 TSval=902191062 TSecr=2266837547
8	5.521064	127.0.0.1	127.0.0.1	TCP	57	8080 → 50214 [PSH, ACK] Seq=1 Ack=4 Win=488256 Len=1 TSval=902191062 TSecr=2266837547 [TCP segment of a reassembled PDU]
9	5.521082	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=4 Ack=2 Win=488256 Len=0 TSval=2266837547 TSecr=902191062
10	11.708354	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [FIN, ACK] Seq=4 Ack=2 Win=488256 Len=0 TSval=2266843734 TSecr=902191062
11	11.708407	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [ACK] Seq=2 Ack=5 Win=488256 Len=0 TSval=902197249 TSecr=2266843734
12	11.708456	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [FIN, ACK] Seq=2 Ack=5 Win=488256 Len=0 TSval=902197249 TSecr=2266843734
13	11.708500	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=5 Ack=3 Win=488256 Len=0 TSval=2266843734 TSecr=902197249

Figur 3: wireshark - tcp

## Three way handshake:

Three-way handshake brukes i TCP/IP-nettverk for å etablere en pålitelig forbindelse mellom en klient og en server. Den består av tre trinn:

**SYN:** Klienten sender en SYN (synchronize) melding til serveren for å starte en forbindelse og angir en sekvensnummer for å starte sekvenseringen av meldingene.

**SYN-ACK:** Serveren svarer med en SYN-ACK (synchronize-acknowledge) melding. Denne meldingen bekrefter mottak av klientens SYN og sender sitt eget sekvensnummer som en del av handshake-prosessen.

**ACK:** Klienten sender en ACK (acknowledge) melding tilbake for å bekrefte mottak av serverens SYN-ACK melding, igjen ved å øke sekvensnummeret med én.

Etter denne prosessen er forbindelsen etablert, og data kan begynne å overføres mellom klienten og serveren. Three-way handshake sikrer at begge parter er klare for dataoverføring og bidrar til å synkronisere sekvensnumrene som brukes for å holde styr på dataene som sendes.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	68	50214 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=2266832026 TSecr=0 SACK_PERM
2	0.000066	127.0.0.1	127.0.0.1	TCP	68	8080 → 50214 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=902185541 TSecr=2266832026 SACK_PERM
3	0.000075	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2266832026 TSecr=902185541
4	0.000082	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8080 → 50214 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=902185541 TSecr=2266832026
6	5.520982	127.0.0.1	127.0.0.1	TCP	59	50214 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=408256 Len=3 TSval=2266837547 TSecr=902185541 [TCP segment of a reassembled PDU]
7	5.521013	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [ACK] Seq=1 Ack=4 Win=408256 Len=0 TSval=902191062 TSecr=2266837547
8	5.521064	127.0.0.1	127.0.0.1	TCP	57	8080 → 50214 [PSH, ACK] Seq=1 Ack=4 Win=408256 Len=1 TSval=902191062 TSecr=2266837547 [TCP segment of a reassembled PDU]
9	5.521082	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=4 Ack=2 Win=408256 Len=0 TSval=2266837547 TSecr=902191062
10	11.708354	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [FIN, ACK] Seq=4 Ack=2 Win=408256 Len=0 TSval=2266843734 TSecr=902191062
11	11.708407	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [ACK] Seq=2 Ack=5 Win=408256 Len=0 TSval=902197249 TSecr=2266843734
12	11.708456	127.0.0.1	127.0.0.1	TCP	56	8080 → 50214 [FIN, ACK] Seq=2 Ack=5 Win=408256 Len=0 TSval=902197249 TSecr=2266843734
13	11.708500	127.0.0.1	127.0.0.1	TCP	56	50214 → 8080 [ACK] Seq=5 Ack=3 Win=408256 Len=0 TSval=2266843734 TSecr=902197249

Nr	Source / destination	protocol	Port	Info	Seq, Act
1	127.0.0.1	TCP	50214 - 8080	SYN	0,-
2	127.0.0.1	TCP	8080 - 50214	SYN, ACT	0,1
3	127.0.0.1	TCP	50214 - 8080	ACT	1,1

Pakke Nr 4 (TCP window update) indikerer at serveren har ledig plass i bufferen og kan ta imot mer data fra klienten.

En kan også se «PSH, ACK» dette er regneoperasjonene.

#### Dataoverføring

Nr	Source / Destination n	protocol	Port	Info	Seq, Act
6	127.0.0.1	TCP	50214 - 8080	PSH, ACK	1,1
7	127.0.0.1	TCP	8080 - 50214	ACK	1,1
8	127.0.0.1	TCP	50214 - 8080	PSH, ACK	1,4
9	127.0.0.1	TCP	8080 - 50214	ACK	1,4

Pakke 5: Serveren gir data til klienten. PSH-flagget viser at serveren ber operativsystemet om å sende dataen til mottakerapplikasjonen. ACK-flagget anerkjenner den siste mottatte pakken fra klienten.

Pakke 6: Klienten har mottatt dataene som ble sendt i pakke 5. Bekreftelsesnummeret 1 indikerer at klienten har mottatt data opp til byte 1 og forventer byte 1 som neste byte.

#### Tilkoblingen avsluttes

Nr	Source / Destination n	protocol	Port	Info	Seq, Act
10	127.0.0.1	TCP	50214 - 8080	FIN, ACK	4,2
11	127.0.0.1	TCP	8080 - 50214	ACK	2,5
12	127.0.0.1	TCP	50214 - 8080	FIN, ACK	2,5
13	127.0.0.1	TCP	8080 - 50214	ACK	5,3

Klienten sender til slutt «FIN, ACK». Dette er klientens måte å initialisere termineringen av koblingen. Dette betyr at den er ferdig å sende data

webserver

Koden starter en enkel webserver som lytter på port 3000. Ved oppstart forsøker den å åpne nettleseren din automatisk til adressen «http://localhost:3000». Serveren oppretter en socket for å lytte etter innkommende forbindelser, konfigurerer den for gjenbruk av adresse/port, og binder den til den angitte porten på alle tilgjengelige nettverksinterfjes.

Når serveren aksepterer en innkommende forbindelse, leser den HTTP-forespørselen fra klienten og bygger en HTTP-respons. Responsen inneholder en statuslinje for å indikere at forespørselen ble vellykket behandlet, en content-type header for å spesifisere at innholdet er HTML, og en enkel HTML-side. HTML-siden viser en velkomstmelding og lister opp hver linje i HTTP-forespørselen som klienten sendte, formatert som en punktliste.

Til slutt sender serveren denne responsen tilbake til klienten, lukker klientforbindelsen, og avslutter ved å lukke sin egen lytte-socket. Denne prosessen demonstrerer hvordan en server kan akseptere en forbindelse, behandle en forespørsel, og sende et svar tilbake til klienten.

```

#define PORT 3000

int main() {
    system("open http://localhost:3000");

    int server_fd, client_socket;
    struct sockaddr_in address;
    char buffer[1024] = {0};

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    int opt = 1; // Definir en variabel for å holde verdien 1
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    bind(server_fd, (struct sockaddr *)&address, sizeof(address));
    listen(server_fd, 3);

    client_socket = accept(server_fd, NULL, NULL);

    read(client_socket, buffer, 1024);

    std::string response = "HTTP/1.0 200 OK\nContent-Type: text/html; charset=utf-8\n\n";
    response += "<HTML><BODY>";
    response += "<H1>HELLO! You have connected to my simple web-server:</H1>";
    response += "<h3>Header from clients is:</h3><UL>";

    std::stringstream requestHeaders(&buffer);
    std::string line;
    while (std::getline(&requestHeaders, &line) && line != "\n") {
        response += "<LI>" + line + "</LI>";
    }
    response += "</UL></BODY></HTML>";

    send(client_socket, response.c_str(), response.length(), 0);
    close(client_socket);
    close(server_fd);

    return 0;
}

```

Figur 4: webserver

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	:::1	:::1	TCP	88	50536 → 3000 [SYN, ACK] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=4064592117 TSecr=0 SACK_PERM
2	0.000059	:::1	:::1	TCP	64	3000 → 50536 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
3	0.000222	127.0.0.1	127.0.0.1	TCP	68	50537 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1991528029 TSecr=0 SACK_PERM
4	0.000376	127.0.0.1	127.0.0.1	TCP	68	3000 → 50537 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1216372502 TSecr=1991528029 SACK_PERM
5	0.000391	127.0.0.1	127.0.0.1	TCP	56	50537 → 3000 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1991528029 TSecr=2116372502
6	0.000401	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 3000 → 50537 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1216372502 TSecr=1991528029
7	0.000654	127.0.0.1	127.0.0.1	HTTP	826	GET / HTTP/1.1
8	0.000732	127.0.0.1	127.0.0.1	TCP	56	3000 → 50537 [ACK] Seq=1 Ack=771 Win=407488 Len=0 TSval=1216372502 TSecr=1991528029
9	0.001088	127.0.0.1	127.0.0.1	TCP	1135	3000 → 50537 [PSH, ACK] Seq=1 Ack=771 Win=407488 Len=1079 TSval=1216372504 TSecr=1991528029 [TCP segment of a reassembled PDU
10	0.001136	127.0.0.1	127.0.0.1	TCP	56	50537 → 3000 [ACK] Seq=771 Ack=1080 Win=407168 Len=0 TSval=1991528031 TSecr=1216372504
11	0.001185	127.0.0.1	127.0.0.1	HTTP	56	HTTP/1.0 200 OK (text/html)
12	0.001232	127.0.0.1	127.0.0.1	TCP	56	50537 → 3000 [ACK] Seq=771 Ack=1081 Win=407168 Len=0 TSval=1991528031 TSecr=1216372504
13	0.052633	127.0.0.1	127.0.0.1	TCP	56	50537 → 3000 [FIN, ACK] Seq=771 Ack=1081 Win=407168 Len=0 TSval=1991528082 TSecr=1216372504
14	0.052673	127.0.0.1	127.0.0.1	TCP	56	3000 → 50537 [ACK] Seq=1081 Ack=772 Win=407488 Len=0 TSval=1216372555 TSecr=1991528082
15	0.129792	:::1	:::1	TCP	88	50538 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=2694948735 TSecr=0 SACK_PERM
16	0.129829	:::1	:::1	TCP	64	3000 → 50538 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
17	0.129920	127.0.0.1	127.0.0.1	TCP	68	50539 → 3000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=4024842441 TSecr=0 SACK_PERM
18	0.129944	127.0.0.1	127.0.0.1	TCP	44	3000 → 50539 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figur 5: wireframe – http

Nr	Source / destination	protocol	Port	Info	Seq, Act
3	127.0.0.1	TCP	50537 - 8080	SYN	0,-
4	127.0.0.1	TCP	8080 - 50537	SYN, ACT	0,1



5	127.0.0.1	TCP	50537 - 8080	ACT	1,1
---	-----------	-----	--------------	-----	-----

## http

Nr	Source / Destination	protocol	Info	
7	127.0.0.1	HTTP	GET / HTTP/1.1	
11	127.0.0.1	HTTP	HTTP/1.1 200 OK	

Pakke 7: Inneholder en HTTP GET-forespørsel fra klienten til serveren for å hente hovedsiden «/».

Pakke 11: Dette er serverens HTTP 200 OK-respons på GET-forespørselen i pakke 7, som indikerer at forespørselen var vellykket og at det blir returnert en HTML-fil.

## Øving 4

### Oppgaven

Denne oppgaven består av to deler. I den første delen av oppgaven, skaper du en kalkulatorapplikasjon der en klient sender beregningsforespørsler til en server ved hjelp av UDP, en forbindelsesløs protokoll kjent for sin enkelhet og effektivitet for enkel dataoverføring. Denne delen fokuserer på grunnleggende nettverksprogrammering og er en øvelse i å sende og motta data over nettverk uten en etablert forbindelse.

I den andre delen av oppgaven, dykker du dypere inn i nettverkssikkerhet ved å sette opp en server som bruker TLS/SSL, protokoller designet for å sikre kryptert kommunikasjon over internett.

UDP kalkulator  
programkode  
client:

```

7 ▶ int main() {
8     int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
9     if (sock == -1) {
10         std::cerr << "Socket creation failed" << std::endl;
11         return -1;
12     }
13
14     struct sockaddr_in serv_addr;
15     serv_addr.sin_family = AF_INET;
16     serv_addr.sin_port = htons(9000);
17
18     if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
19         std::cerr << "Invalid address" << std::endl;
20         close(sock);
21         return -1;
22     }
23
24     while (true) {
25         char operation;
26         int num1, num2;
27
28         std::cout << "Enter your first number: ";
29         std::cin >> num1;
30         std::cout << "Enter your second number: ";
31         std::cin >> num2;
32         std::cout << "Enter the operation (+, -, *, /): ";
33         std::cin >> operation;
34
35         std::stringstream ss;
36         ss << operation << " " << num1 << " " << num2;
37         std::string message = ss.str();
38
39         sendto(sock, message.c_str(), message.length(), 0, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
40
41         char buffer[1024] = {0};
42         socklen_t serv_addr_size = sizeof(serv_addr);
43         int bytes_received = recvfrom(sock, buffer, sizeof(buffer) - 1, 0, (struct sockaddr*)&serv_addr, &serv_addr_size);
44         if (bytes_received == -1) {
45             std::cerr << "recvfrom failed" << std::endl;
46             continue;
47         }
48
49         std::cout << "Answer from server: " << buffer << std::endl;
50
51         std::string decision;
52         std::cout << "Do you want to continue? (y/n): ";
53         std::cin >> decision;
54         if (decision != "y" && decision != "Y") {
55             break;
56         }
57     }
58
59     close(sock);
60     return 0;
61 }

```

Figur 6: client (p4)

Klienten er en del av kalkulatorapplikasjon som kommuniserer med en server via UDP. Ved oppstart forsøker klienten å opprette en UDP-socket. Hvis dette mislykkes, avsluttes programmet med en feilmelding.

Klienten konfigurerer deretter serverens adresseinformasjon, inkludert IP-adresse og portnummer 9000, for å vite hvor meldingene skal sendes. Ved hjelp av en løkke, lar klienten brukeren kontinuerlig utføre matematiske operasjoner. For hver iterasjon, ber den brukeren

om to tall og hvilken matematisk operasjon som skal utføres på disse tallene. Brukeren kan velge mellom addisjon, subtraksjon, multiplikasjon, og divisjon

Når brukeren har oppgitt input, konverteres denne informasjonen til en streng og sendes til serveren ved hjelp av `sendto`-funksjonen. Klienten venter deretter på et svar fra serveren ved å bruke `recvfrom`-funksjonen, som blokkerer inntil et svar mottas eller en feil oppstår. Svaret, som forventes å være resultatet av den matematiske operasjonen, skrives ut til brukeren.

Etter å ha mottatt og vist svaret, spør klienten brukeren om de ønsker å utføre en ny beregning. Hvis brukeren svarer nei, avsluttes løkken, og klienten lukker socketen før programmet avsluttes. Dette tillater brukeren å utføre så mange beregninger som ønsket med serveren før programmet avsluttes på en kontrollert måte.

### Server:

```
6
7 void handle_client(const std::string& message, struct sockaddr_in client_addr, int server_socket) {
8     char operation;
9     int num1, num2, result = 0;
10    std::istringstream ss( message);
11    ss >> operation >> num1 >> num2;
12
13    std::string response;
14    switch (operation) {
15        case '+':
16            result = num1 + num2;
17            response = "" + std::to_string( result);
18            break;
19        case '-':
20            result = num1 - num2;
21            response = "" + std::to_string( result);
22            break;
23        case '*':
24            result = num1 * num2;
25            response = "" + std::to_string( result);
26            break;
27        case '/':
28            if (num2 == 0) {
29                response = "Error: Division by zero";
30            } else {
31                result = num1 / num2;
32                response = "" + std::to_string( result);
33            }
34            break;
35        default:
36            response = "Error: Invalid operation";
37            break;
38    }
39
40    sendto(server_socket, response.c_str(), response.length(), 0, (struct sockaddr*)&client_addr, sizeof(client_addr));
41 }
42
```

```

43 int main() {
44     int server_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
45     if (server_socket == -1) {
46         std::cerr << "Socket creation failed" << std::endl;
47         return -1;
48     }
49
50     struct sockaddr_in server_addr;
51     int port = 9000;
52     server_addr.sin_family = AF_INET;
53     server_addr.sin_addr.s_addr = INADDR_ANY;
54     server_addr.sin_port = htons(port);
55
56     if (bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
57         std::cerr << "Bind failed" << std::endl;
58         close(server_socket);
59         return -1;
60     }
61
62     std::cout << "Server listening on port " << port << std::endl;
63
64     char buffer[1024];
65     struct sockaddr_in client_addr;
66     socklen_t client_addr_size = sizeof(client_addr);
67
68     while (true) {
69         memset(&buffer, 0, sizeof(buffer));
70         int bytes_received = recvfrom(server_socket, buffer, sizeof(buffer), 0, (struct sockaddr*)&client_addr, &client_addr_size);
71         if (bytes_received == -1) {
72             std::cerr << "Error in recvfrom" << std::endl;
73             continue;
74         }
75
76         std::string message(&buffer, bytes_received);
77         handle_client(message, client_addr, server_socket);
78     }
79
80     close(server_socket);
81     return 0;

```

Figur 7: server (p4)

Ved oppstart forsøker serveren å opprette en UDP-socket og binder den til portnummer 9000 for å lytte etter innkommende forespørsler på hvilken som helst interface ('INADDR\_ANY'). Hvis opprettelsen eller bindingen av socketen mislykkes, avsluttes programmet med en feilmelding.

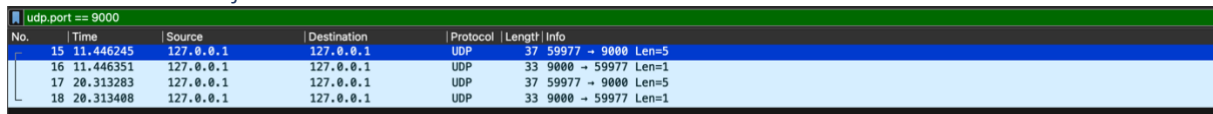
Når serveren er klar og lytter på porten, går den inn i en løkke hvor den venter på forespørsler fra klienter. For hver forespørsel leser serveren meldingen som inneholder to tall og en indikator av symbolene +, -, \*, eller /.

Serveren dekodeer meldingen ved hjelp av en 'std::istream' for å ekstrahere operasjonen og de to tallene. Basert på operasjonen som er forespurt av klienten, utfører serveren den tilsvarende matematiske operasjonen. Hvis operasjonen er gyldig, beregnes resultatet, og serveren genererer en respons som inneholder resultatet av beregningen. For divisjon kontrollerer serveren også om divisor er null for å unngå divisjon med null. Hvis operasjonen ikke er gyldig, eller det er en annen feil, genereres en feilmelding som respons.

Til slutt sender serveren responsen tilbake til klientens adresse ved å bruke 'sendto'-funksjonen, som også spesifiserer klientens adresse og portnummer for å sikre at svaret blir sendt tilbake til riktig mottaker. Dette gjøres inne i 'handle\_client'-funksjonen, som tar den

mottatte meldingen, klientens adresseinformasjon, og serverens socket som argumenter for å behandle forespørselen og sende tilbake et svar.

## UDP kommunikasjon



No.	Time	Source	Destination	Protocol	Length	Info
15	11.446245	127.0.0.1	127.0.0.1	UDP	37	59977 → 9000 Len=5
16	11.446351	127.0.0.1	127.0.0.1	UDP	33	9000 → 59977 Len=1
17	20.313283	127.0.0.1	127.0.0.1	UDP	37	59977 → 9000 Len=5
18	20.313408	127.0.0.1	127.0.0.1	UDP	33	9000 → 59977 Len=1

Figur 8: Wireshark – udp

UDP, som står for User Datagram Protocol, er en enkel, forbindelsesløs nettverksprotokoll som brukes for å sende datagrammer over et IP-nettverk uten å garantere levering, rekkefølge eller feilfri overføring av pakker. Dette gjør UDP til et raskt og effektivt valg for applikasjoner som krever høy ytelse og kan tolerere noe tap av data, som strømming av video eller lyd, spill, eller noen ganger, sanntidskommunikasjon. UDP tillater applikasjoner å sende meldinger, kalt datagrammer, til andre vertsprogrammer på et IP-nettverk med minimal overhead, da det ikke oppretter en forbindelse før data sendes og ikke utfører feilkontroll etter at data er sendt.

Nr	Source / destination	protocol	Port	Info
15	127.0.0.1	UDP	59977 - 9000	LEN = 5
16	127.0.0.1	UDP	9000 - 59977	LEN = 1
17	127.0.0.1	UDP	59977 - 9000	LEN = 5
18	127.0.0.1	UDP	9000 - 59977	LEN = 1

## TLS

### Programkode

#### JavaSSLServer:

koden implementerer en enkel SSL-sikret server som lytter etter innkommende tilkoblinger på port 8000. Serveren bruker Java sin `SSLServerSocketFactory` for å opprette en SSL-beskyttet `ServerSocket`, som sikrer at all data som sendes mellom serveren og klientene er kryptert og sikker.

Når serveren er startet, skriver den ut en melding om at den lytter etter tilkoblinger, og venter deretter på at en klient skal koble seg til. Når en klient kobler seg til serveren, aksepterer den tilkoblingen og oppretter en `Socket` for å kommunisere med klienten.

For å håndtere dataoverføring, oppretter serveren en `PrintWriter` og en `BufferedReader`. `PrintWriter` brukes til å sende data tilbake til klienten, mens `BufferedReader` leser data som kommer fra klienten. Serveren leser linjer med tekst fra klienten og skriver ut disse

linjene til konsollen. Deretter sender den samme linjen tilbake til klienten som en ekko-respons.

Kommunikasjonen fortsetter i en løkke hvor serveren leser og ekkoer linjer med tekst til den ikke mottar flere data, hvorpå den avslutter tilkoblingen. Til slutt, skriver serveren ut en melding om at forbindelsen er lukket.

```
3 import java.io.BufferedReader;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
6 import java.io.PrintWriter;
7 import java.net.ServerSocket;
8 import java.net.Socket;
9 import java.util.logging.Level;
10 import java.util.logging.Logger;
11 import javax.net.ssl.SSLServerSocketFactory;
12
13 /**
14  * @web http://java-buddy.blogspot.com/
15  */
16  * erik
17  *
18  * public class JavaSSLServer {
19  *
20  *     1 usage
21  *     static final int port = 8000;
22  *
23  *     2 erik
24  *     public static void main(String[] args) {
25  *
26  *         SSLServerSocketFactory sslServerSocketFactory =
27  *             (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
28  *
29  *         try {
30  *             ServerSocket sslServerSocket =
31  *                 sslServerSocketFactory.createServerSocket(port);
32  *             System.out.println("SSL ServerSocket started");
33  *             System.out.println(sslServerSocket.toString());
34  *
35  *             Socket socket = sslServerSocket.accept();
36  *             System.out.println("ServerSocket accepted");
37  *
38  *             PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
39  *             try (BufferedReader bufferedReader =
40  *                 new BufferedReader(
41  *                     new InputStreamReader(socket.getInputStream()))) {
42  *                 String line;
43  *                 while((line = bufferedReader.readLine()) != null){
44  *                     System.out.println(line);
45  *                     out.println(line);
46  *                 }
47  *             }
48  *             System.out.println("Closed");
49  *
50  *         } catch (IOException ex) {
51  *             Logger.getLogger(JavaSSLServer.class.getName())
52  *                 .log(Level.SEVERE, msg: null, ex);
53  *         }
54  *     }
55  * }
```

Figur 9: javaSSLServer

## JavaSSLClient:

Koden implementerer en enkel SSL-klient som kobler seg til en server på port 8000 ved hjelp av SSL for en sikret kommunikasjon. Klienten bruker `SSLSocketFactory` for å opprette en SSL-sikret `Socket` forbindelse til serveren, som sikrer at data som utveksles mellom klienten og serveren er kryptert og beskyttet mot avlytting eller datainnblanding.

Når tilkoblingen er etablert, oppretter klienten en `PrintWriter` for å sende data til serveren og en `BufferedReader` for å motta data fra serveren. Programmet går deretter inn i en løkke hvor det ber brukeren om å skrive inn tekst via konsollen. Denne teksten sendes til serveren, og klienten venter på et svar. Når svaret mottas, vises det for brukeren. Dette fortsetter til brukeren skriver inn "q", som er signalet for å avslutte løkken og dermed også programmet.

```
/**
 * @web http://java-buddy.blogspot.com/
 */
@erik
public class JavaSSLClient {

    1 usage
    static final int port = 8000;

    @erik
    public static void main(String[] args) {

        SSLSocketFactory sslSocketFactory =
            (SSLSocketFactory)SSLSocketFactory.getDefault();
        try {
            Socket socket = sslSocketFactory.createSocket("localhost", port);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
            try (BufferedReader bufferedReader =
                new BufferedReader(
                    new InputStreamReader(socket.getInputStream()))) {
                Scanner scanner = new Scanner(System.in);
                while(true){
                    System.out.println("Enter something:");
                    String inputLine = scanner.nextLine();
                    if(inputLine.equals("q")){
                        break;
                    }

                    out.println(inputLine);
                    System.out.println(bufferedReader.readLine());
                }
            }
        } catch (IOException ex) {
            Logger.getLogger(JavaSSLClient.class.getName())
                .log(Level.SEVERE, msg: null, ex);
        }
    }
}
```

Figur 10: javaSSLClient



## Lagene:

I kommunikasjonen mellom SSL/TLS-sikrede servere og klienter benyttes flere lag av nettverksprotokoller for å sikre en pålitelig og sikker dataoverføring. På transportlaget, opererer SSL eller TLS for å legge til et lag av sikkerhet ved å kryptere dataene som sendes over nettverket.

På nettverkslaget benyttes IP for å adressere og rute pakker fra sender til mottaker.

## Kommunikasjon

No.	Time	Source	Destination	Protocol	Length	Info
466	1871.886283	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1947485199 TSecr=0 SACK_PERM
467	1871.886367	127.0.0.1	127.0.0.1	TCP	68	8000 → 51143 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=776916881 TSecr=1947485199 SACK_PERM
468	1871.886376	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1947485199 TSecr=776916881
469	1871.886380	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8000 → 51143 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=776916881 TSecr=1947485199
470	1875.691865	127.0.0.1	127.0.0.1	TLSv1..	496	Client Hello
471	1875.691120	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=1 Ack=441 Win=407808 Len=0 TSval=776919965 TSecr=1947489083
472	1875.728312	127.0.0.1	127.0.0.1	TLSv1..	183	Server Hello
473	1875.728366	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=441 Ack=128 Win=408128 Len=0 TSval=1947489113 TSecr=776919995
474	1875.729706	127.0.0.1	127.0.0.1	TLSv1..	62	Change Cipher Spec
475	1875.729801	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=441 Ack=134 Win=408128 Len=0 TSval=1947489122 TSecr=776920004
476	1875.732946	127.0.0.1	127.0.0.1	TLSv1..	62	Change Cipher Spec
477	1875.732999	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=134 Ack=447 Win=407808 Len=0 TSval=776920007 TSecr=1947489125
478	1875.735276	127.0.0.1	127.0.0.1	TLSv1..	126	Application Data
479	1875.735328	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=447 Ack=204 Win=408064 Len=0 TSval=1947489127 TSecr=776920009
480	1875.743529	127.0.0.1	127.0.0.1	TLSv1..	997	Application Data
481	1875.743594	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=447 Ack=1145 Win=407104 Len=0 TSval=1947489136 TSecr=776920018
482	1875.769424	127.0.0.1	127.0.0.1	TLSv1..	358	Application Data
483	1875.769473	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=447 Ack=1447 Win=406848 Len=0 TSval=1947489162 TSecr=776920044
484	1875.778845	127.0.0.1	127.0.0.1	TLSv1..	146	Application Data
485	1875.778895	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=447 Ack=1537 Win=406720 Len=0 TSval=1947489163 TSecr=776920045
486	1875.774630	127.0.0.1	127.0.0.1	TLSv1..	146	Application Data
487	1875.774683	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=1537 Ack=537 Win=407744 Len=0 TSval=776920049 TSecr=1947489167
488	1875.775644	127.0.0.1	127.0.0.1	TLSv1..	98	Application Data
489	1875.775664	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=1537 Ack=579 Win=407680 Len=0 TSval=776920050 TSecr=1947489168
490	1875.779055	127.0.0.1	127.0.0.1	TLSv1..	1261	Application Data
491	1875.779109	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=579 Ack=2742 Win=405504 Len=0 TSval=1947489171 TSecr=776920053
492	1875.779648	127.0.0.1	127.0.0.1	TLSv1..	98	Application Data
493	1875.779665	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=579 Ack=2784 Win=405504 Len=0 TSval=1947489171 TSecr=776920053
494	1879.014513	127.0.0.1	127.0.0.1	TLSv1..	98	Application Data
495	1879.014546	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=2784 Ack=621 Win=407680 Len=0 TSval=776923289 TSecr=1947492487
496	1879.015045	127.0.0.1	127.0.0.1	TLSv1..	98	Application Data
497	1879.015095	127.0.0.1	127.0.0.1	TCP	56	51143 → 8000 [ACK] Seq=621 Ack=2826 Win=405440 Len=0 TSval=1947492487 TSecr=776923289

Figur 11: wireshark ssl

Nr	Source / destination	protocol	Port	Info	Seq, Act
466	127.0.0.1	TCP	51143 - 8000	SYN	0,-
467	127.0.0.1	TCP	8000 - 51143	SYN, ACT	0,1
468	127.0.0.1	TCP	51143 - 8000	ACT	1,1

Etter tilkoblingen med three way handshake blir det sendt «hello client», deretter «hello server». Med dette foreslår klienten SSL/TLS protokoller og “cipher suites”. Med Server Hello velger serveren hvilke protokoller og cipher suite som faktisk skal bli brukt.

En kan også se det står «Change Cipher Spec». Dette er en melding brukt i protokollene for å indikere at partene i en kommunikasjon nå vil begynne å bruke de avtalte sikkerhetsinnstillingene for kryptering av meldinger. Denne meldingen sendes som en del av håndtrykksprosessen for å signalisere overgangen fra usikret til sikret kommunikasjon ved å aktivere den valgte krypteringsalgoritmen.

## Nedkobling

No.	Time	Source	Destination	Protocol	Length	Info
630	2577.106536	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [ACK] Seq=2826 Ack=622 Win=407680 Len=0 TSval=777621369 TSecr=1948190487
631	2577.111913	127.0.0.1	127.0.0.1	TLSv1..	96	Application Data
632	2577.111956	127.0.0.1	127.0.0.1	TCP	56	8000 → 51143 [FIN, ACK] Seq=2866 Ack=622 Win=407680 Len=0 TSval=777621374 TSecr=1948190487
633	2577.111957	127.0.0.1	127.0.0.1	TCP	44	51143 → 8000 [RST] Seq=622 Win=0 Len=0

Figur 12: nedkobling av ssl



Nr	Source / destination	protocol	Port	Info	Seq, Act
630	127.0.0.1	TCP	8000 - 51143	SYN	2826,622
631	127.0.0.1	TLSv1.3		SYN, ACT	
632	127.0.0.1	TCP	8000 - 51143	ACT	22866,62
633	127.0.0.1	TCP	51143 - 8000	RST	622

TLS- versjon	TLSV1.3(version 1.2)
Hvilken kryptografisuite som velges	TLS_AES_256_GCM_SHA384

## Tjenerens sertifikat

Sertifikatet under har et unikt fingeravtrykk og signatur som er unikt for dette sertifikatet.

Når du bruker kommandoen `keytool -list -v -keystore examplestore`, listes det opp alle nøklene og sertifikatene som er lagret, inkludert informasjon som alias for hver oppføring, sertifikatets eier og utsteder, serienummer, gyldighetsperiode, og den offentlige nøkkelen. Sertifikatet bruker en 2048-bits RSA nøkkel

```
erik@Erik-sin-MBP del2 % keytool -list -v -keystore examplestore
Enter keystore password:
Keystore type: PKCS12
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: signfiles
Creation date: 22. feb. 2024
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Erik turmo, OU=NTNU, O=" NTNU", L=trondheim, ST=trøndelag, C=47
Issuer: CN=Erik turmo, OU=NTNU, O=" NTNU", L=trondheim, ST=trøndelag, C=47
Serial number: 3cdac5f64635385
Valid from: Thu Feb 22 22:54:21 CET 2024 until: Fri Feb 21 22:54:21 CET 2025
Certificate fingerprints:
    SHA1: 4E:AE:E7:3D:9C:0F:E6:91:8B:FD:7D:03:08:62:67:85:A0:4E:16:D7
    SHA256: 7C:98:20:F5:BB:44:56:DA:67:C3:09:62:79:A1:50:20:41:62:A3:7A:48:6D:A4:1E:4B:56:15:C1:D9:A8:DE:74
Signature algorithm name: SHA384withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
```

## Kryptografisuite

krypteringssuiten som brukes er «TLS\_AES\_256\_GCM\_SHA384»

Krypteringsuiten "TLS\_AES\_256\_GCM\_SHA384" beskriver en sett med algoritmer som brukes for å sikre kommunikasjon i en TLS (Transport Layer Security) sesjon. Den gir detaljert informasjon om hvilke krypterings- og hashalgoritmer som anvendes for å beskytte dataene som utveksles.

AES\_256\_GCM: Dette står for Advanced Encryption Standard med en nøkkellengde på 256 bit, som brukes for kryptering av meldingsinnholdet. GCM er en modus for operasjonen som tilbyr både autentisering og konfidensialitet (kryptering), og er kjent for sin effektivitet og sikkerhet.

SHA384: Dette refererer til bruk av SHA-384 for å generere en meldingsautentiseringskode. Dette sikrer integriteten og autentisiteten til dataene som sendes, ved å beskytte mot uautoriserte endringer.

Samlet sett indikerer "TLS\_AES\_256\_GCM\_SHA384" at krypteringsuiten bruker AES med en 256-bits nøkkel i GCM-modus for å kryptere data, og SHA-384 for å sikre integriteten og autentisiteten til meldingene.

#### Hvordan sesjonsnøkler opprettes

Sesjonsnøkler opprettes gjennom en prosess kjent som nøkkelutveksling, som er en del av håndtrykket mellom to parter i en kryptert kommunikasjon, for eksempel mellom en nettleser og en webserver over TLS. Denne prosessen starter når partene først blir enige om en krypteringsalgoritme og deretter bruker en sikker metode for å dele eller etablere en hemmelig nøkkel. Under nøkkelutvekslingen genereres offentlige og private nøkler; den private nøkkelen holdes hemmelig, mens den offentlige nøkkelen deles med den andre parten. Ved hjelp av disse nøklene kan partene beregne en delt hemmelighet uten å faktisk overføre den over nettverket. Denne delte hemmeligheten, sammen med tilleggsinformasjon som er unik for hver sesjon, brukes deretter til å generere sesjonsnøkler. Sesjonsnøklerne er midlertidige og brukes til å kryptere og dekryptere data som sendes i den aktuelle kommunikasjonssesjonen, sikrende at overført informasjon forblir konfidensiell og uforandret under overføringen.