IDATT2104 - Datakom Oblig 1

Jakob Grønhaug (jakobkg@stud.ntnu.no)

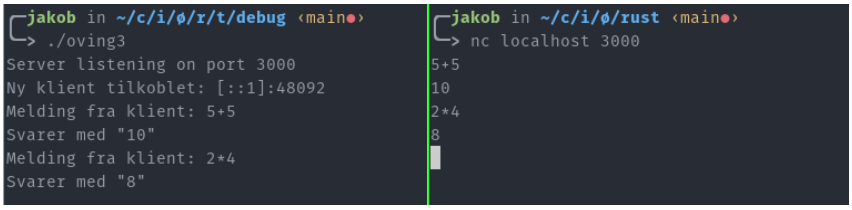
21. februar 2023

Innhold

1 P3: Sockets, TCP, HTTP og tråder

1.1 Enkel tjener/klient

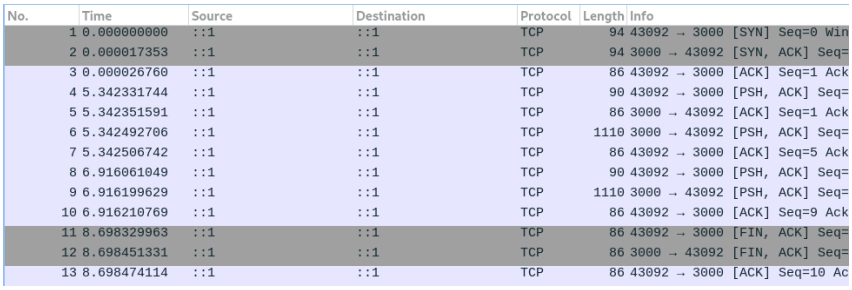
Til denne oppgaven har jeg skrevet en enkel tjener som lytter på port 3000 etter koblinger fra klienter, og svarer på enkle regnestykker. Med Netcat som klient ser interaksjon med denne tjeneren slik ut:



Figur 1: Enkel tjener som utfører regnestykker, brukt med Netcat som klient

Dette tjener-programmet er basert på `TcpListener`-klassen fra Rusts standardbibliotek[1], som oppretter en **TCP**-kobling over IP. Når programmet kun kjører lokalt hos meg er det spesifikt IPv6 som brukes, som kan sees i pakkefangsten i følgende figur (adresse `::1` i stedet for `127.0.0.1`). Siden dette programmet bruker TCP foregår det et handshake (SYN, SYN/ACK, ACK) når en klient kobles til tjeneren, og et lignende handshake (FIN, FIN/ACK, ACK) når koblingen lukkes fordi enten klienten eller tjeneren avsluttes. Dette kan også sees i en pakkefangst med Wireshark:

[1] https://doc.rust-lang.org/std/net/struct.TcpListener.html



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	::1	::1	TCP	94	43092 → 3000 [SYN] Seq=0 Win
2	0.000017353	::1	::1	TCP	94	3000 → 43092 [SYN, ACK] Seq=
3	0.000026760	::1	::1	TCP	86	43092 → 3000 [ACK] Seq=1 Ack
4	5.342331744	::1	::1	TCP	90	43092 → 3000 [PSH, ACK] Seq=
5	5.342351591	::1	::1	TCP	86	3000 → 43092 [ACK] Seq=1 Ack
6	5.342492706	::1	::1	TCP	1110	3000 → 43092 [PSH, ACK] Seq=
7	5.342506742	::1	::1	TCP	86	43092 → 3000 [ACK] Seq=5 Ack
8	6.916061049	::1	::1	TCP	90	43092 → 3000 [PSH, ACK] Seq=
9	6.916199629	::1	::1	TCP	1110	3000 → 43092 [PSH, ACK] Seq=
10	6.916210769	::1	::1	TCP	86	43092 → 3000 [ACK] Seq=9 Ack
11	8.698329963	::1	::1	TCP	86	43092 → 3000 [FIN, ACK] Seq=
12	8.698451331	::1	::1	TCP	86	3000 → 43092 [FIN, ACK] Seq=
13	8.698474114	::1	::1	TCP	86	43092 → 3000 [ACK] Seq=10 Ac

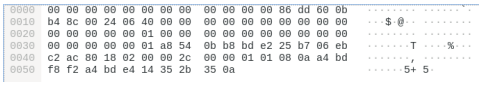
Figur 2: Pakkefangst av tjener med kun en klient

Siden koblingen bruker TCP er all kommunisjon gjort i parvise pakker, der den ene pakken er at en av partene sender data i en pakke og den andre parten (dersom alt går som det skal) svarer med en ACK-pakke for å bekrefte at dataen ble mottatt. I dette programmet er kommunikasjonen på formen regnestykke/svar, så forventet kommunikasjon for et enkelt regnestykke er klient-data→tjener-ack→tjener-data→klient-ack. Dette kan for eksempel sees i figur 2, der pakkene 4-7 viser en slik frem-og-tilbake mellom klienten og tjeneren.

Merk også at svaret fra tjeneren er betydelig større enn forespørselen fra klienten. Dette skyldes at jeg på tjener-siden oppretter en buffer på 1kB (1024 byte), putter dataen som tjeneren ønsker å sende inn i denne bufferen og sender hele bufferen av gårde. En bedre løsning ville vært å kun sende den delen av bufferen som faktisk inneholder data, men den løsningen som er brukt her fungerer også selv om den har enorm overhead. Den relevante delen av koden der denne bufferen opprettes og sendes uten å sjekke hvor mye av bufferen som faktisk er nyttig data er:

```
1    // Tømmer respons-bufferen før behandling
2    responsebuf = [0; 1024];
3
4    // Behandler forespørselen og genererer svar vha funksjonsobjektet response_fn
         som legges i responsebuf
5    (self.response_fn)(message.into_owned(), &mut responsebuf);
6
7    // Sender responsebuf tilbake, avslutter om sendingen feiler av noen grunn
8    if let Err(e) = socket.write_all(&responsebuf).await {
9        eprintln!("Problem ved skrivning til socket: {e:?}");
10       return;
11   }
```

Merk spesielt linje 2 der bufferen fylles med 1024 bytes med `0x00`, og linje 8 der denne bufferen sendes av gårde gjennom nettverket uten at den forkortes fra sin fulle størrelse på 1024 bytes på noen måte. På klient-siden er Netcat flinkere til å ikke sende en drøss med unødvendig data, forskjellen kan sees veldig tydelig i pakkevisningen i Wireshark, se figur 3 nedenfor.



(a) En pakke sendt fra Netcat med data "5+5"

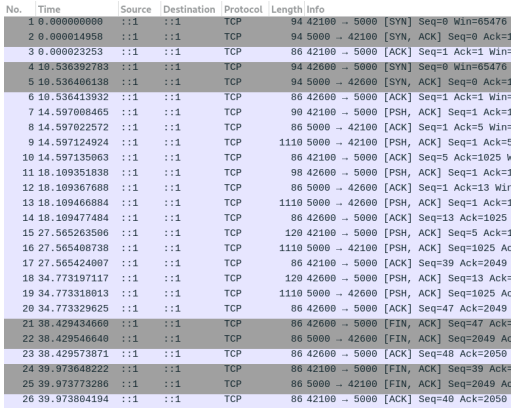
(b) En pakke sendt fra tjeneren med data "10", fulgt av i overkant av tusen null-bytes

Figur 3: Sammenligninger av pakker sendt fra Netcat og fra tjener-programmet

1.2 Flere samtidige klienter

For å utvide tjener-programmet har jeg brukt Rust-biblioteket Tokio[2] for å enklere opprette en tråd for hver klient som kobles til tjeneren. På denne måten kan flere klienter kommunisere med tjeneren til samme tid uten å måtte vente på tur eller oppleve nevneverdig ventetid fra tjeneren mottar en forespørsel til den faktisk behandles. Det er fortsatt Rusts `TcpListener` og de tilhørende typene som ligger i bunnen av programmet, og dermed fortsatt TCP som benyttes. I følgende skjermbilde fra Wireshark kan man se to separate SYN→SYN/ACK→ACK-sekvenser og to separate FIN→FIN/ACK→ACK-sekvenser, for hhv. oppkobling og nedkobling mellom klient og tjener.

(a) Terminal med en tjener og to klienter



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	::1	::1	TCP	94	42100 → 5000 [SYN] Seq=0 Win=65476
2	0.000014958	::1	::1	TCP	94	5000 → 42100 [SYN, ACK] Seq=0 Ack=1
3	0.000023253	::1	::1	TCP	86	42100 → 5000 [ACK] Seq=1 Ack=1 Win=
4	10.536392783	::1	::1	TCP	94	42600 → 5000 [SYN] Seq=0 Win=65476
5	10.536406138	::1	::1	TCP	94	5000 → 42600 [SYN, ACK] Seq=0 Ack=1
6	10.536413932	::1	::1	TCP	86	42600 → 5000 [ACK] Seq=1 Ack=1 Win=
7	14.597008465	::1	::1	TCP	90	42100 → 5000 [PSH, ACK] Seq=1 Ack=1
8	14.597022572	::1	::1	TCP	86	5000 → 42100 [ACK] Seq=1 Ack=5 Win=
9	14.597124924	::1	::1	TCP	1110	5000 → 42100 [PSH, ACK] Seq=1 Ack=5
10	14.597135063	::1	::1	TCP	86	42100 → 5000 [ACK] Seq=5 Ack=1025 W
11	18.109351838	::1	::1	TCP	98	42600 → 5000 [PSH, ACK] Seq=1 Ack=1
12	18.109367688	::1	::1	TCP	86	5000 → 42600 [ACK] Seq=1 Ack=13 Wir
13	18.109466884	::1	::1	TCP	1110	5000 → 42600 [PSH, ACK] Seq=1 Ack=1
14	18.109477484	::1	::1	TCP	86	42600 → 5000 [ACK] Seq=13 Ack=1025
15	27.565263506	::1	::1	TCP	120	42100 → 5000 [PSH, ACK] Seq=5 Ack=1
16	27.565408738	::1	::1	TCP	1110	5000 → 42100 [PSH, ACK] Seq=1025 Ac
17	27.565424007	::1	::1	TCP	86	42100 → 5000 [ACK] Seq=39 Ack=2049
18	34.773197117	::1	::1	TCP	120	42600 → 5000 [PSH, ACK] Seq=13 Ack=
19	34.773318013	::1	::1	TCP	1110	5000 → 42600 [PSH, ACK] Seq=1025 Ac
20	34.773329625	::1	::1	TCP	86	42600 → 5000 [ACK] Seq=47 Ack=2049
21	38.429434660	::1	::1	TCP	86	42600 → 5000 [FIN, ACK] Seq=47 Ack=
22	38.429546640	::1	::1	TCP	86	5000 → 42600 [FIN, ACK] Seq=2049 Ac
23	38.429573871	::1	::1	TCP	86	42600 → 5000 [ACK] Seq=48 Ack=2050
24	39.973648222	::1	::1	TCP	86	42100 → 5000 [FIN, ACK] Seq=39 Ack=
25	39.973773286	::1	::1	TCP	86	5000 → 42100 [FIN, ACK] Seq=2049 Ac
26	39.973804194	::1	::1	TCP	86	42100 → 5000 [ACK] Seq=40 Ack=2050

(b) Pakkefangst av en tjener og to klienter som gjør oppkobling, noen forespørsler hver, og nedkobling

Figur 4: Kjøring og pakkefangst av en tjener med to samtidige klienter

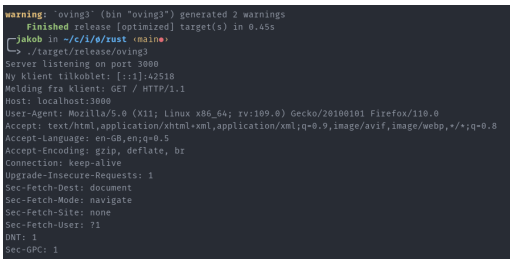
I denne varianten av tjeneren er koden som behandler forespørsler og genererer svar endret fra å lese og utføre regnestykker til å bare kopiere dataen fra den mottatte forespørselen til bufferen som sendes i retur (en ekko-tjener). Selve koden som behandler sending av responser fra tjeneren til klientene er den samme som i den forrige oppgaven, så problemet med overdrevent store svar-pakker opptrer her også på samme måte som før (se f.eks. pakke

[2]https://docs.rs/tokio/1.25.0/tokio/

9 og 13). Den eneste forskjellen i håndteringen av klienter er at det nå opprettes en ny tråd for hver klient, som håndterer sin klientens forespørsler. Dette er oppnådd ved å pakke inn den relevante koden i en `loop {...}`, og bruke `tokio::spawn(...)` til å opprette en ny tråd hver gang en oppkoblingsforespørsel mottas.

1.3 Web med nettleser som klient

Takket være litt lur modularisering av tjener-programmet er all koden som behandler oppkobling og samtidighet gjenbrukt fra forrige oppgave, og en ny funksjon som leser HTTP-forespørsler og genererer et HTML-dokument som spesifisert i oppgaveteksten er alt som trengs av ny kode. Denne nye funksjonen behandler alle innkommende forespørsler likt og anerkjenner ikke muligheten for at noen skulle kunne ønske seg noe annet enn et HTML-dokument i svar på forespørselen sin, men fungerer likefullt mer enn godt nok til denne oppgaven. Siden all kode som har med nettverk/sockets å gjøre er gjenbrukt er det fremdeles TCP som er i bruk. Det er verdt å merke seg at siden det ikke er noen kryptering eller sikkerhet som SSL eller TLS inne i bildet kan denne tjeneren kun HTTP, ikke HTTPS.



(a) Terminal med web-tjener

(b) Svaret på forespørselen fra (a), vist i Firefox

21	20.423656365	::1	::1	TCP	94	49366 → 8080 [SYN] Seq=0 Win=65476 Len
22	20.423673898	::1	::1	TCP	94	8080 → 49366 [SYN, ACK] Seq=0 Ack=1 Wi
23	20.423684198	::1	::1	TCP	86	49366 → 8080 [ACK] Seq=1 Ack=1 Win=655
24	20.423763717	::1	::1	HTTP	542	GET / HTTP/1.1
25	20.423773695	::1	::1	TCP	86	8080 → 49366 [ACK] Seq=1 Ack=457 Win=6
26	20.423954785	::1	::1	TCP	1110	8080 → 49366 [PSH, ACK] Seq=1 Ack=457
27	20.423966046	::1	::1	TCP	86	49366 → 8080 [ACK] Seq=457 Ack=1025 Wi
28	20.423985904	::1	::1	HTTP	86	HTTP/1.0 200 OK (text/html)
29	20.424092213	::1	::1	TCP	86	49366 → 8080 [FIN, ACK] Seq=457 Ack=10
30	20.424106760	::1	::1	TCP	86	8080 → 49366 [ACK] Seq=1026 Ack=458 Wi

(c) Pakkene som går frem og tilbake mellom nettleser og webtjener. Pakker fra port 49366 stammer fra nettleseren, og pakker fra port 8080 stammer fra tjeneren.

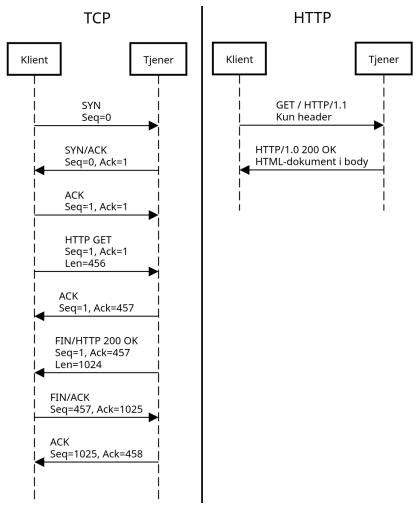
Figur 5: Web-tjener, klient og pakkefangst

Denne gangen er det implementert HTTP, en protokoll som Wireshark faktisk forstår og kan vise noe mer informasjon om! De tidligere oppgavene har kun vært tekst over TCP uten noen ordentlig protokoll, som begrenser hvor mye metadata og nyttig info Wireshark kan vise om pakkene som går frem og tilbake. Nå som det er gyldige HTTP-pakker som sendes kan Wireshark vise litt mer info basert på protokollen. Se f.eks. pakkefangsten i figuren

over, der pakke 24 og 28 er HTTP-pakker med hhv `GET / HTTP/1.1` og `HTTP/1.0 200 OK` med respons-body av typen `text/html`. All denne infoen Wireshark fremhever er hentet fra headerne til pakkene som sendes frem og tilbake.

Siden deler av HTTP er implementert kan vi nå se på denne pakkeutvekslingen i flere lag, og se at det er en betydelig enklere interaksjon som foregår på applikasjonslaget (HTTP) enn på transportlaget (TCP). Der transportlaget håndterer oppkobling, nedkobling og synkronisering, trenger applikasjonslaget bare å sende meldinger frem og tilbake mellom klient og tjener uten å måtte dobbeltsjekke om trafikken faktisk kommer frem eller ikke.

Det er denne typen abstraksjoner som gjør lagmodellen for nettverk så nyttig, det at man på applikasjonslaget kan arbeide med en antagelse om at de underliggende lagene i modellen gjør jobben sin uten at man trenger å tenke på dem



Figur 6: Sekvensdiagram som viser den samme trafikken slik den ser ut på transportlaget og på applikasjonslaget

2 P4: UDP, TLS

2.1 UDP-kalkulator

I denne oppgaven er kalkulator-programmet fra tidligere endret til å kommunisere over UDP i stedet for TCP. Dette fjerner en betydelig andel overhead fra kommunikasjonen og forenkler programmeringen noe, siden koblingen mellom klient og tjener ikke lenger er tilstandsfull og dermed ikke trenger handshakes, oppkobling eller nedkobling for å fungere. Data bare sendes av gårde, og programmet trenger ikke lenger ta stilling til om den blir mottatt og kvittert hos mottakeren.

Nok en gang er en klasse fra Rusts standardbibliotek tatt i bruk for å behandle kommunikasjon, denne gangen `UdpSocket`.