Datanet
Obligatorisk opgave 2: TCP

René Hansen
Michael Nilou
Anders Bjerg Pedersen
Hold 1

19. september 2007

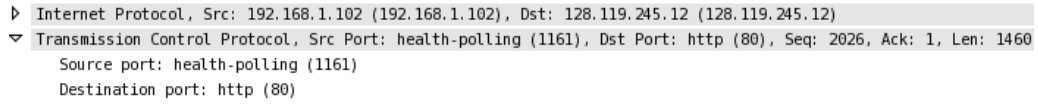
Indledning

Denne opgave går ud på at analysere TCPs måde at transmittere og retransmittere pakker over internettet. Undervejs vil vi studere begreber som timeouts, flow control, congestion control og acknowledgements.

Wireshark TCP

2. A first look at the captured trace

Q1: Vi finder klientens IP-adresse og portnummer i en af TCP-pakkerne (se screenshot nedenfor):
`Src: 192.168.1.102 (192.168.11.102), Src Port: 1161`.

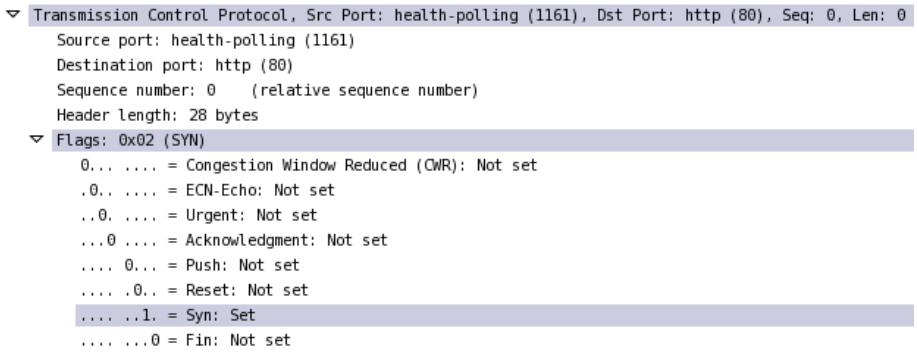


Q2: Samme sted som i Q1 finder vi tilsvarende oplysninger for serveren (se samme screenshot):
`Dest: 128.119.245.12 (128.119.245.12), Dest Port: www (80)`.

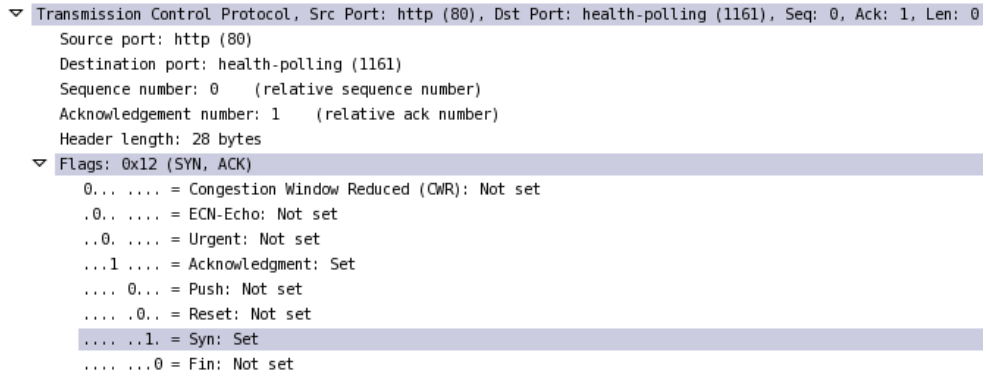
Q3: Vi har ikke lavet eget trace.

3. TCP Basics

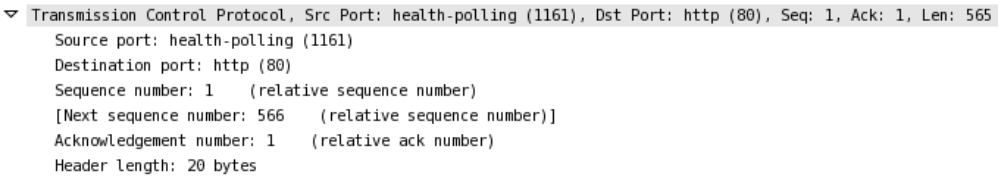
Q4: Af nedenstående screenshot ser vi, at følgende gælder:
`Sequence number: 0`, `Flags: 0x02 (SYN), .... ..1. = Syn: Set`.



Q5: Af nedenstående screenshot af SYNACK-pakken fås følgende:
`Sequence number: 0`, `Acknowledgement number: 1`.
Acknowledgement number er tallet for den næste byte der forventes indlæst. SYNACK-sekvensnummeret er SYN-nummeret + 1, dvs. SYNACK-sekvensnummeret er i vores tilfælde 1.
Segmentet er et SYNACK-segment, fordi både SYN- og ACK-bitten i Flags er sat.



Q6: Sekvensnummeret er `Sequence number: 1` (serveren forventer stadig at modtage den første byte af eventuel data).



Q7: Følgende data er hentet fra pakkeoversigten i Wireshark. EstimatedRTT er beregnet ud fra formlen i bogen:

Sequence #	**Time sent**	**Time ACK**	**RTT**	**EstimatedRTT**
1	0.026477	0.053937	0.02746	0.02746
566	0.041737	0.053937	0.0122	0.00393
2026	0.054026	0.077294	0.023268	0.00635
3486	0.054690	0.124085	0.069125	0.01420
4946	0.077405	0.169118	0.091713	0.02389
6406	0.078157	0.217299	0.139142	0.03830

Q8: Følgende informationer er kopieret fra TCP-delen af pakkerne i Wireshark:

Header	**Data**	**I alt**
Header length: 20 bytes	Data (565 bytes)	585
Header length: 20 bytes	Data (1460 bytes)	1480
Header length: 20 bytes	Data (1460 bytes)	1480
Header length: 20 bytes	Data (1460 bytes)	1480
Header length: 20 bytes	Data (1460 bytes)	1480
Header length: 20 bytes	Data (1460 bytes)	1480

Q9: Vores minimum buffer size er 5840 bytes (fra pakke 2, `Est. RTT`). Forbindelsen bliver ikke "throttled"undervejs, da vinduet ikke formindskes under transmissionen af pakkerne (det stiger støt, indtil det når 62780 bytes). Desuden er flaget "`Congestion Window Reduced`"ikke sat på nogen af pakkerne (søgningen "`tcp.flags.cwr == 1`"giver et tomt resultat)
(`0... .... = Congestion Window Reduced (CWR): Not set`).

Q10: Nej, der bliver ikke retransmitteret nogen pakker. Sorterer man på "`ip.src == 192.168.1.102`"(afsenderens IP-adresse) i Wireshark, kan man desuden se, at afsenderpakkernes sekvensnumre er strengt voksende.

Q11: Der bliver oftest ACK'et for 1460 bytes (standarddatastørrelsen på datasegmentet i en TCP-pakke). Dette kan ses ved at observere, at ACK-numrene stiger med 1460 hele tiden. Vi får i starten en ACK på hver pakke, dvs. vi ikke umiddelbart oplever steder, hvor der ACK'es kumuleret for flere pakker ad gangen. Når vi når op til omkring pakke 50, bliver der kun ACK'et for hver anden pakke (der sendes 6 pakker, ACK'es for 3...).

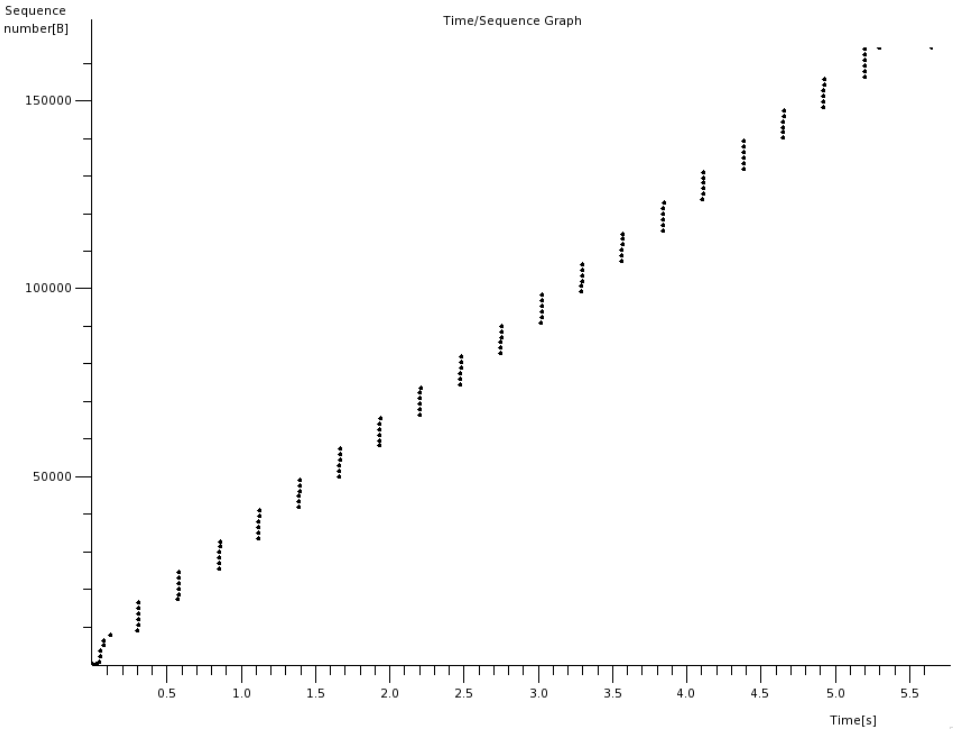
Q12: Vi anvender formlen fra side 287 i bogen (4. udgave). Her er $MSS = 1460$bytes, og vores RTT estimerer vi ved hjælp af Wireshark (Statistics > TCP Stream Graph > Round Trip Time Graph) til $RTT \simeq 0.19$s. Da vi ikke mister nogen pakker undervejs, er vores $L = 1.0$. Sætter vi ind i formelen, får vi:

$$\text{avg. throughput } = \frac{1.22 \cdot MSS}{RTT \cdot \sqrt{L}} = \frac{1.22 \cdot 1460bytes}{0.19s \cdot \sqrt{1.0}} = 9374.7368\frac{bytes}{s} = 73.24\frac{kbit}{s}.$$

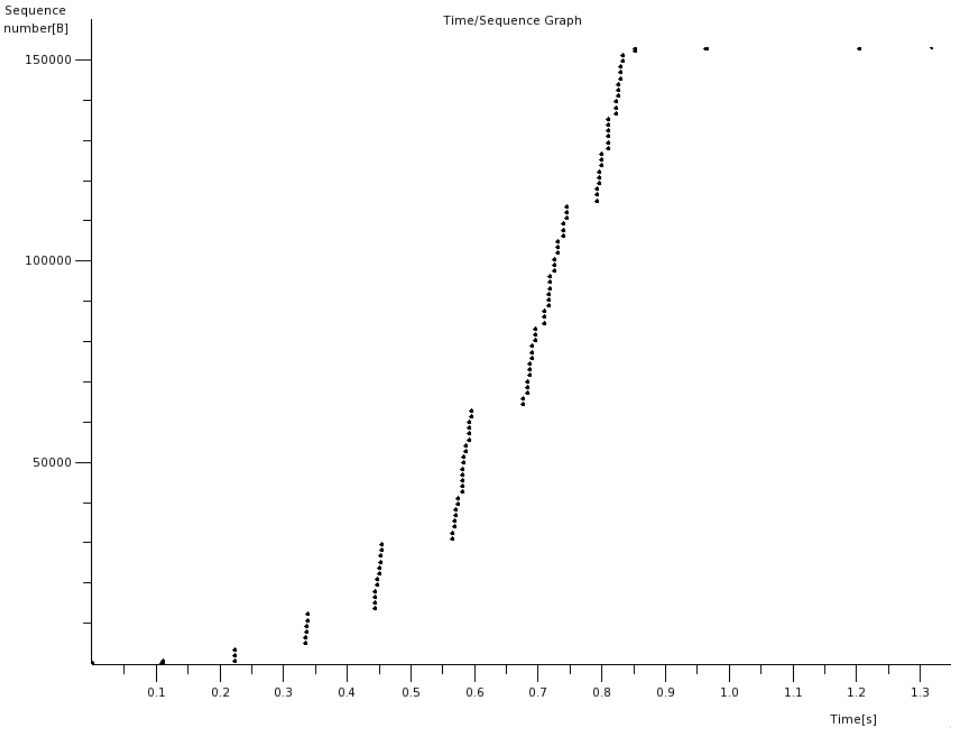
I denne udregning er der taget visse forbehold. Formlen i bogen beregner kun throughput for ren data i TCP, den medtager f.eks. ikke headers i TCP eller de andre lag.

4. TCP congestion control in action

Q13: Herunder ses Stevens-grafen for tracet. Som man kan ane på grafens begyndelse, stiger antallet af pakker per tidsenhed eksponentielt i starten, hvorefter antallet stiger lineært. Der startes altså med en "Slow Start". Dog kan vi ikke umiddelbart forklare, hvorfor TCP-protokollen ikke forsøger at sende flere pakker end 6 ad gangen. Vi må derfor gætte på, at send-bufferen hos klienten er for lille til at sende mere end 6 pakker ad gangen. Grunden til, at den kan nå at sende 7 pakker første gang, kunne være, at første pakke kun er 565 bytes, hvor resten er 1460 bytes.



Q14: Herunder ses Stevens-grafen for vores egen kørsel med Wireshark. Man kan nu tydeligt se, hvordan antallet af pakker fordobles hver gang, men at vi ikke når grænsen for antal pakker, før vi er færdige med at sende filen.



Ekstraopgave

I det følgende mener vi, når vi siger "vi"i teksten, *klienten* og ikke serveren.

Q1: Umiddelbart ser det ud, som om TCP-protokollen reagerer som forventet. PC'en i vores ende reagerer ligeledes som forventet, altså må vi antage, at problemet ligger enten mellem de to hosts eller på DIKUs maskine. ACKs fra serveren er meget langsomme om at komme frem (hvis de overhovedet kommer), eller de er meget få i antal. Dette ses bl.a. på den timeout, der sker mellem pakkerne 14 og 15, hvor pakke 15 bliver sendt igen.

Q2: I pakke 7 sender vi en ACK på pakkerne 2, 4 og 6 til serveren. I pakke 8 sendes så en ny pakke, som der svares på i pakke 9.

I pakke 13 ACK'er vi for alle bytes op til og med 400, dvs. for pakkerne 2, 4, 6, 9 og 12. I pakke 14 sender vi en ny pakke (400-480), der opnår en timeout, hvorved vi i pakke 15 sender den samme pakke (400-480) igen, plus en ekstra (480-560).

I pakke 17 ACK'er serveren for alt op til og med 480. I pakke 18 sender vi endnu en pakke (800-880), som timer ud, hvorefter vi i pakke 19 sender endnu mere (800-960). Vi bemærker, at vores timeouts bliver længere og længere, eftersom serveren bliver langsommere og langsommere til at svare.

I pakke 27 bekræfter vi over for serveren, at vi har sendt den 960 bytes og modtaget 1056 bytes fra den. I pakke 28 sender vi en ny pakke (960-1040), som timer ud, derfor sender vi pakken igen i pakke 29, nu med to ekstra pakker vedhæftet (960-1280).

I pakke 47 forsøges en sidste gang at sende pakke 34 igen, men uden held. Forbindelsen har nu efter al sandsynlighed nået sin ultimative timeout, og derfor genoprettes forbindelsen igen i pakke 48 (der bliver gensendt som pakke 49). I de efterfølgende pakker "forhandles"TCP-forbindelsen igen, hvorefter SSH-forbindelsen genoprettes fra pakke 52 og frem.

Q3: Der er flere muligheder for fejlkilder. Dog springer et par af dem i øjnene. Hvis netværket er overbelastet, vil der kunne opstå pakketab, som kan forårsage situationen umiddelbart før pakke 48. Vi kan også konstatere, at de pakker, der når frem, er i orden (er ikke ødelagt), hvilket ikke umiddelbart tyder på deciderede fejl mellem de to hosts. Dog vil det være oplagt, hvis der blev skabt congestion mellem de to hosts, f.eks. i en router, der så ville medføre, at pakkerne ankom sent (i klumper). Generelt er det da også klient-siden, der i starten venter på serveren, mens billedet vender, når vi kommer op i nærheden af pakkerne 35-45.

Q4: Et oplagt forslag til videre fejlfinding ville være, at Klaus tog sin computer med ind på DIKU og kørte en trace, mens han forsøgte at genskabe problemet. Hvis Klaus ikke oplever problemet på sin egen maskine på DIKU, må problemet ligge i Klaus' forbindelse. Problemet kan ikke ligge på serveren, da Klaus åbenbart er den

eneste, der oplever problemet (dvs. andre kan uden problemer udføre handlingerne hjemmefra).

Andre muligheder kunne være at prøve en eller flere traceroutes mellem Klaus og DIKU, og efterfølgende ping'e de forskellige hubs mellem vores to hosts, for muligvis at kunne isolere problemet til en af routerne på nettet. Til sidst vil man så pinge DIKU, hvis der ikke er fundet problemer undervejs.

Man vil også forsøge med andre programmer end f.eks. pine, der bruger en SSH-tunnel til DIKU for at se, om disse programmer opfører sig normalt. Dette vil enten kunne udelukke eller bekræfte, at problemet ligger i pine.

Konklusion

Vi har i denne opgave set, hvordan TCP håndterer protokollens forskellige egenskaber, såsom retransmission, flow control og acknowledgements. Vi har også analyseret en mere kompliceret problemstilling, hvor det ikke umiddelbart var muligt at komme frem til en entydig løsning ud fra Wiresharks analyser af TCP-protokollens opførsel.