Studie av beräknings- och nätverksprestanda i 10GbE HPC-kluster

Av

Carl Lind

Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden

Abstract

This thesis has been performed at Saab Technologies in Gothenburg. Saab is a global defense- and security company that specializes in air-, land and marine defense. Saab in Gothenburg develops effective solutions for monitoring, warning, location and defense systems.

The thesis work which has been initiated by Saab, involves development of methods and tools in order to dimension the network components in an Ethernet communicating HPC-Cluster (High performance computing). The cluster's task is to perform signal processing for radar solutions.

During the thesis work different network protocols have been evaluated in order to evaluate which protocol that proves most suitable to use for signal transmission between the radar and the cluster. It turned out that many protocols are developed or depend on the two most common protocols in the transport layer, TCP and UDP. These protocols where therefore examined more thoroughly by testing them against each other. The goal was to find out which protocol that had the upper hand by measuring their performance. This means to find out how fast both protocols can send data before unacceptable packet loss occur for the application and under those circumstances find out the packet delay as well. This cannot be done by theoretical calculations but has to be tested practically. First by testing on one computer, then between two computers and finally on the HPC-Cluster.

The result of the thesis is a number of graphs which show the two protocol's difference in performance.

Keywords: HPC-Cluster, Network performance, TCP, UDP, Measurement.

Sammanfattning

Detta examensarbete har utförts på Saab Technologies i Göteborg. Saab är ett globalt försvars- och säkerhetsföretag verksamt inom bland annat flyg-, land- och marinförsvar. Saab i Göteborg inriktar sig på att bygga effektiva lösningar för övervakning, förvarning, lokalisering och skydd.

Examensarbetet, som har utformats av Saab, går ut på att ta fram metoder och verktyg för att dimensionera nätverksdelarna i ett ethernet-kommunicerande HPC-kluster (High Performance Computing). Klustrets uppgift är att utföra signalbehandling för radar.

Under examensarbetet har olika nätverksprotokoll utvärderats för att man ska kunna ta fram vilket protokoll som är lämpligast att använda för signalöverföring mellan radar och klustret. Det visade sig att många protokoll bygger på eller beror på de två vanligaste protokollen i transportlagret, nämligen UDP och TCP. Dessa protokoll har därför utvärderats mer noggrant genom att testas mot varandra. Syftet är att ta reda på vilket protokoll som är bäst genom att mäta prestanda. Detta innebär att ta reda på hur snabbt de båda protokollen kan skicka data innan, för applikationen, oacceptabla paketförluster inträffar och hur fördröjningen blir. Detta går inte att få fram teoretiskt utan måste testas i praktiken. Först genom att testa på en dator, sedan mellan två datorer och slutligen på klustret.

Resultatet på examensarbetet är ett antal grafer som visar skillnaden mellan de två protokollens prestanda.

Nyckelord: HPC-Kluster, Nätverksprestanda, TCP, UDP, Mätning.

Förord

Detta examensarbete har utförts på Saab Technologies i Göteborg under vårterminen 2015. Examensarbetet är en avslutande del av högskoleingenjörsutbildningen i datateknik vid Lunds Tekniska Högskola.

Jag skulle vilja tacka Lovisa Björklund för möjligheten att utföra detta examensarbete på Saab samt min handledare Mattias Bucht för all hjälp och vägledning.

Jag vill även tacka min examinator Christian Nyberg på LTH samt min handledare på LTH, Mats Lilja, för all positiv feedback och hjälp med strukturen av arbetet.

Innehållsförteckning

1. Inledning

Detta avsnitt ger en kortfattad inblick i vad arbetet går ut på och vilka frågeställningar som ska besvaras. Det klargörs även vilka avgränsningar som fått göras.

1.1. Bakgrund

Examensarbetet har utförts på SAAB technologies i Göteborg. Saab är ett globalt försvars- och säkerhetsföretag verksamt inom bland annat flyg-, land- och marinförsvar. Saab i Göteborg inriktar sig på att bygga effektiva lösningar för övervakning, förvarning, lokalisering och skydd. Affärsområdet avser radar och elektronisk krigsföring som täcker flyg-, land- och marina radartillämpningar.

Examensarbetet går ut på att ta fram metoder och verktyg för att dimensionera nätverksdelarna i ett ethernetkommunicerande HPC-kluster (high performance computing) som utför signalbehandling för radar. Det som har föranlett detta arbete är framförallt att 10GBe (10 Gigabit ethernet) HPC-kluster nu är tillgängliga även för industriella och militära tillämpningar. HPC-kluster systemen har funnits i över ett årtionde, men en tillämpning av dem i försvarssammanhang har inte funnits tidigare. Att använda 10GBe HPC-kluster som en beräkningsnod för radartillämpningar är därför lämpligt. Anledningen till att det hittills inte har tillämpats är för att militärtillämpningar ställer mycket höga krav på hårdvaran, till exempel att den ska klara mycket låga temperaturer, vibrationer och elektromagnetisk strålning.

Data skickas från ett antal radarelement över en länk till en switch som sedan fördelar all data till ett antal processorkort. Dessa processorkort är belägna i ett rack, d.v.s. en hylla där ett antal kort får plats. Mätningarna för en radar görs i intervall och för varje intervall kommer radarns antennelement att ha uppmätt signaler som ska behandlas. Dessa signaler görs om till en digital videosignal som sedan fördelas och beräknas av HPC-klustret.

1.2. Syfte

Syftet med examensarbetet är att ta fram en metod för dimensionering av nätverket i HPC-klustret samt fördelning av beräkningslasten på olika processorkort, givet en problemstorlek (indata) och ett beräkningsproblem. Detta innebär att man bör ta reda på vilket protokoll som är lämpligast att använda i HPC-klustrets nätverk och hur data ska delas upp. Uppdelningsproblemet innebär att man på ett effektivt sätt delar upp data till de olika processorkorten i HPC-klustret. Med data eller indata syftar man i huvudsak på signaler från ett radarsvep. Overhead är dels överlappning av data och dels paketstorlek beroende på icke användningsbar data eller header i datapaket. Se kapitel 2.5 om uppdelningsproblemet. Syftet med resultatet är att tala om hur man ska ställa in klustret för att uppnå så bra prestanda som möjligt.

1.3. Problemformulering

Ett antal radarsignaler skickas till en switch som sedan fördelar signalerna till ett antal processorkort. Signalerna skickas med ett visst intervall och för varje intervall ska denna data behandlas. Detta ger följande funktion:

(latens, paketförlust, beräkningsbelastning) = f(signalbandbredd, antal processorkort, typ av nätverksprotokoll).

Med latens menas här den totala fördröjningen, tiden det tar att skicka data över en länk eller att beräkna den. Med länk avses förbindelsen mellan sändare och mottagare. Beräkningsbelastning är hur belastad hårdvaran och till viss del operativsystemet blir under intensiv beräkning. Signalbandbredd är hur mycket data man kan både skicka och beräkna på en viss tid. Givet är ett antal indata (radarsignaler) som ger ett visst dataflöde och för dessa vill man kunna få underlag för dimensionering av nätverksklustret. Detta innebär lastfördelning, val av nätverksprotokoll samt antal processorkort, för att få tillräckligt låga värden på vänsterledet i ekvationen ovanför.

Dessa frågeställningar gjordes i början av arbetet:

1. Vilket/vilka nätverksprotokoll är lämpligast för att transportera data i HPC-klustret?
2. Hur många processorkort behövs?
3. Hur mycket overhead kan man som mest tolerera?
4. Vilken metod är lämpligast att använda för att dela upp data mellan de olika processorkorten?
5. Hur mycket användbar data får man plats med på en länk i förhållande till länkens kapacitet?
6. Vilka andra tekniska problem finns med implementationen?

Under arbetets gång har en del frågeställningar utgått och det har även tillkommit två nya frågeställningar. Följande nya frågeställningar har tillkommit:

1. Är det lämpligt att använda stora eller små paketstorlekar för att uppnå hög prestanda?
2. Hur påverkas tiden det tar att skicka ett paket?

1.4. Avgränsningar

Denna rapport fokuserar på att utvärdera ett antal utvalda nätverksprotokoll samt implementera och jämföra UDP mot TCP, för att se vilket som lämpar sig bäst för dataöverföring på klustret. Dessa två protokoll är de vanligast förekommande protokollen i transportlagret och skillnaderna mellan dem är relativt stora. Valet av protokoll är framförallt viktigt eftersom det ger en grund som resten av lösningen kommer att bygga på, när det gäller beräkningar och implementationer.

Det finns även andra protokoll som är tänkbara att användas. Det mest lovande av dem är SCTP, Stream Controlled Transmission Protocol, som dock tidigt har kunnat uteslutas. (se 4.1.3).

De andra protokollen bygger på TCP och UDP vilket leder till att det först är viktigt att ta reda på vilket av dessa två protokoll som är mest lämpligt.

Uppdelningsproblemet har inte hunnits med, dels på grund av begränsad tid men även eftersom implementationen av TCP och UDP visade sig mer omfattande och problematisk än väntat.

Följande frågeställningar har inte kunnat besvaras:

1. Hur många processorkort behövs?
2. Hur mycket overhead kan man som mest tolerera?
3. Vilken metod är lämpligast att använda för att dela upp data mellan de olika processorkorten?

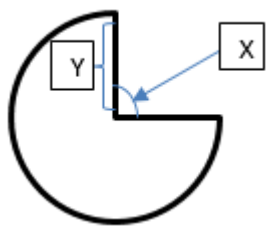
Anledningen till att de här frågeställningarna inte har gått att besvara är för att examensarbetet har fokuserat på att mäta prestandan mellan två noder och inte mellan en till flera noder. Detta har avgränsats på grund av tidsbrist och eftersom uppdelningsproblemet inte har hunnits med.

2. Nulägesbeskrivning

Detta avsnitt beskriver utgångsläget för arbetet och ger en bakgrund till det som arbetet handlar om. Nulägesbeskrivningen beskriver utgångspunkten för arbetet.

2.1. Radar

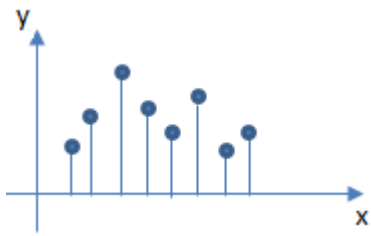
På en radar sitter ett antal antennelement som sänder pulser med jämna intervall. Om pulsen träffar ett hinder så kommer signalen att splittras och reflekteras. En liten mängd av reflektionen kommer då att träffa radarns antennelement. Dessa antennelement är olika i antal beroende på vilken typ av radar man använder. För varje antennelement omvandlas den uppmätta reflektionen till en digital videosignal. En radar som roterar kan representeras med en cirkel som har en mittpunkt (radarn) och en utgångsvinkel där mätningen börjar. Se [24] och [25] samt figur 1 för mer detaljer.



Figur 1. Ett radarsvep sett uppifrån. Y är avståndet från centrum och X är bäring. Bäring är vinkeln mellan utgångspunkten och det siktade föremålet.

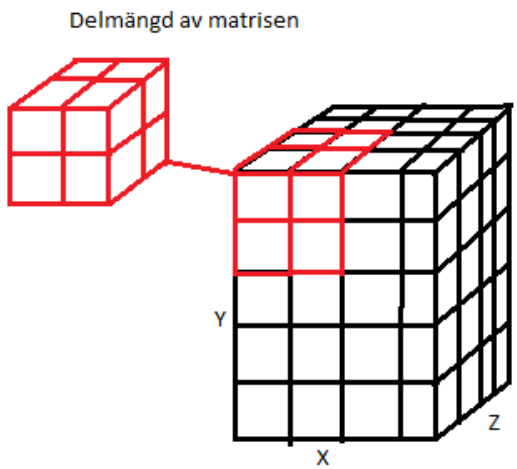
Det grundläggande en radar mäter är två saker, avstånd och riktning. Avståndet mäts med hjälp av RTT/2 (Round-Trip-Time), d.v.s. tiden det tar från att en signal skickas tills att den studsat tillbaks dividerat med 2. Generellt blir det: $\frac{C*t}{2}$. Där c är ljusets hastighet och t är tiden från att pulsen skickats tills reflektionen nått radarn. På en radar sitter antennelementen tätt bredvid varandra i x- och y-led.

Ett hinders elevation (höjd) går då att läsa av från reflektionen i y-led, ett hinder på marknivå ses då som en horisontell reflektion. Målets riktning mäter man med hjälp av bäring. En radar har vanligen norr som sin utgångspunkt och vid den punkten är bäringen 0°. Målets riktning är alltså vinkel X relativt utgångspunkten (norr) i figur 1. Se [24] och [25] för detaljer.



Figur 2. Visar ett koordinatsystem där x är tiden och y är de uppmätta signalernas styrka.

De signaler man får av en radar som snurrar runt kan man illustrera som en tredimensionell matris där y är avståndet från målet, x är bäring och z är numret på ett antennelement på radarn. Se figur 3.



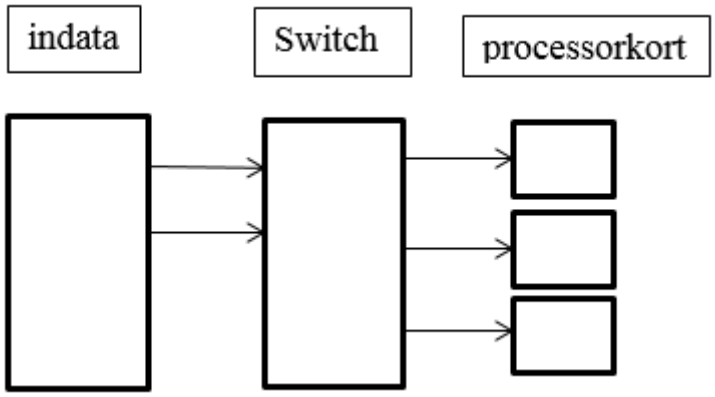
Figur 3. Den röda kuben motsvarar en delmängd av radarmatrisen. Radarmatrisen anses vara den matris man får ut som resultat av ett radarsvep. Varje matriselement innehåller ett tal som representerar signalstyrkan i punkten med bäring x och avstånd y hos antennelement z.

För att man ska kunna behandla radarmatrisen på ett så effektivt sätt som möjligt kan man till exempel använda sig av ett signal-anpassat filter. Detta filter är lika stort som en delmängd av matrisen, se figur 3. Filtret innehåller förutbestämda värden som motsvarar det mål man vill hitta. Filtret förflyttas ett steg i x och y led och summan av filtret multipliceras då med varje element i matrisen. Produkten av varje multiplikation läggs som element i en ny matris. På den plats där värdena är störst i den nya matrisen finns målet som man letar efter. Eftersom radarmatrisen oftast är väldigt stor behöver man dela ner beräkningen i mindre delar, för att det ska blir mer effektivt. Det är HPC-klustrets uppgift att göra denna beräkning. Se [4] samt uppdelningsproblemet i 2.5 för mer detaljer.

2.2. HPC-kluster

Ett HPC-kluster kan se ut på flera olika sätt men principen är samma. Det är en dator med hög beräkningskapacitet som består av ett stort antal datorer eller processorer som samverkar för att utföra stora beräkningar så snabbt som möjligt. (Se [14] för mer information). I detta fall består klustrets beräkningsnoder av processorkort som samverkar för behandla signaler från en radar. Klustret implementeras och körs nästan alltid i Linuxmiljö. HPC-kluster har ett flertal olika tillverkare, bland annat Mellanox Technologies och IBM. Dessa kluster är framförallt tillverkade för att utföra beräkningar på större mängder data men kan också användas för parallell programmering eller datalagring. Både IBM och Mellanox levererar färdigbyggda HPC-kluster efter kundens behov. Ett Processorkort är ett PCI-Expresskort som sitter i en hylla eller rack med flera andra kort. Dessa kort är kopplade till en gemensam switch som fördelar data till de olika korten. Dessa kort har väldigt hög prestanda och är gjorda för att hantera stora mängder data. Se [26] och [27] för mer information om HPC-kluster samt information om Mellanox och IBM.

Data skickas med hjälp av ett visst protokoll över en länk till en switch som sedan fördelar all data.



Figur 4. Ett HPC-kluster.

Man kan kartlägga hela problemet genom att ställa upp följande funktion:

(latens, paketförlust, beräkningsbelastning) = f(signalbandbredd, antal processorkort typ av nätverksprotokoll)

Latens (eng latency), motsvarar all typ av fördröjning. Det är värt att tänka på att det finns vissa restriktioner i form av att ökning på vissa parametrar medför en minskning av andra.

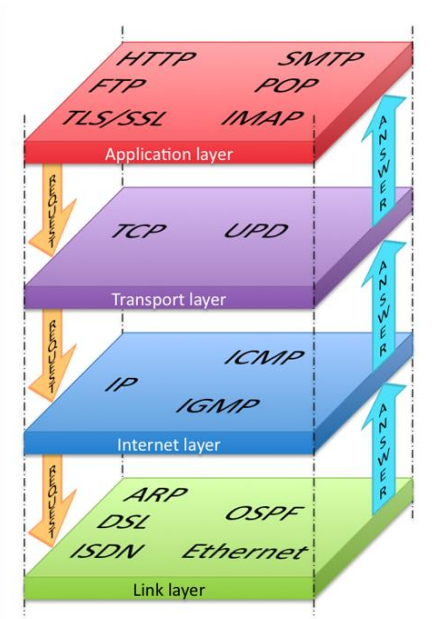
2.3. Nätverksprotokoll

*"A **Protocol** defines the format and the order of the message exchanged between two or more communicating entities as well as the actions taken on the transmission and/or receipt of a message or other event" - [1].*

Även om vi inte reflekterar över det så använder faktiskt vi människor olika protokoll för att kommunicera dagligen. Oftast inleds ett samtal med att man säger hej och sedan förväntar man sig oftast ett hej tillbaks innan man fortsätter prata. Kanske vill man säga något till flera personer eller vill man bara att en person ska höra genom att man viskar.

Principen är samma även för datorer, beroende på informationen och hur den ska förmedlas behöver man veta vilket protokoll som är bäst lämpat för ändamålet.

Internet är uppbyggt av olika lager och varje lager har ett antal protokoll som har olika användningsområden. Alla lager är fördelade i en hierarki där protokollen som ligger överst (applikationsnivå), oftast begränsas eller styrs av de som ligger under. I detta arbete är det viktigt att bedöma vilket protokoll som är lämpligast att använda för att transportera data från en källa till flera mottagare. Utgångspunkten är att bedöma vilket av de båda protokollen Transport Control Protocol, TCP och User Datagram Protocol, UDP som är lämpligast för detta, men även att undersöka vilka alternativ till dessa två som finns och i så fall utvärdera dessa. De båda protokollen är de vanligaste och mest använda protokollen på transportlagernivå enligt [1].



Figur 5. Bild över IP-stacken, bildkälla: [29].

I figur 5 är applikationslagret överst och protokollen på denna nivå ska enbart standardisera kommunikation genom att exempelvis översätta en vanlig domänadress (www.hej.org), till en 32 bitars nätverksadress. Själva kommunikationen mellan två program sköts av transportlagret. Att skicka ett paket eller datagram mellan två noder i ett nätverk görs av nätverkslagret (internet layer i figur 5).

Protokollet som ligger ovanför i hierarkin, exempelvis UDP, säger då till nätverkslagret till vilken adress det ska transportera ett datagram. När datagrammet i sin tur skickas från en nod till en annan i ett nätverk eller över internet används länklagret. En nod avses vara till exempel en router eller en switch i ett nätverk. Ett datagram skickas då ner till länklagret som transporterar paketet från en nod till en annan, för varje nod skickas paketet upp till nätverkslagret igen, som antingen skickar datagrammet vidare (upp i hierarkin), eller ner till länklagret igen beroende på om datagrammet har nått sin destinationsnod. Det går att läsa mer om de olika lagerna i [1].

På transportlagret är det valda protokollets uppgift att transportera ett meddelande eller data från ett program till ett annat program över en länk. Detta görs genom att protokollet kapslar in data i ett paket där en så kallad ”header” läggs på med information enligt [1]. Vilken information som finns i headern beror på vilket protokoll man använder. Detta går dock inte att bestämma såvida man inte tillverkar sitt eget protokoll. När man t.ex. använder UDP och TCP i java går det bara att styra över det som sker på applikationsnivå, allt annat görs av operativsystemet enligt [11]. I java går det även att ändra buffertstorleken, men vad du kan sätta den till beror på inställningarna i operativsystemet. Se 4.1 (teknisk bakgrund) för mer information om varje protokoll som undersökts samt [11].

För att kunna bedöma vilka krav man ska ställa på ett protokoll ställdes följande frågor. Svaren är de krav som SAAB ställer på implementationen.

Hur stor är risken för paketförlust och störningar?

Data skickas över en länk på 50-100m till en switch, vilket innebär att det enbart är **en** nod till nod förbindelse. Denna förbindelse påverkas inte av störningar och så länge paket skickas med ett jämnt flöde som switchen kan hantera, är risken för paketförlust låg.

Är det viktigast att paketen kommer fram i tid eller att protokollet är pålitligt?

Det är ungefär lika viktigt. Paketförlusten får inte vara för stor men det är även viktigt att paketen inte kommer fram för sent. Om det blir en stor paketförlust kommer det orsaka väldigt mycket ”brus” eller störningar som direkt påverkar mätresultatet. Om paket istället kommer fram för sent kan resultatet istället bli inaktuellt på grund av att målet, exempelvis ett flygplan, rör på sig.

Kommer data alltid att skickas i samma takt eller kommer dataflödet att variera?

Trafikflödet är alltid samma, d.v.s. en jämn ström av data kommer att skickas. Radarns svep kommer att vara konstant och data kommer därför att behöva skickas samt beräknas parallellt med mätningen.

Eftersom all data som skickas ska delas upp och beräknas av ett antal processorkort så kan man gå tillväga på två sätt. Här avses startnod vara källan till de data som ska behandlas.

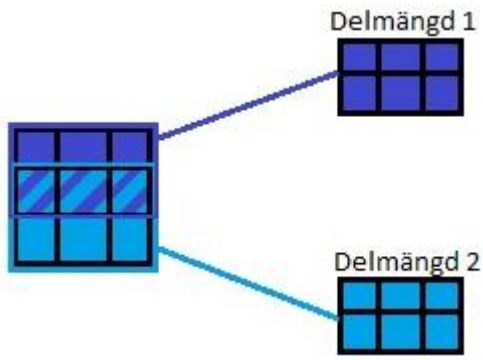
- Om man delar upp all data vid startnoden kan man skicka data parallellt med flera förbindelser, d.v.s. en förbindelse från startnod till varje processorkort i HPC-klustret. Detta gör att switchen enbart behöver skicka vidare trafik till rätt slutnod, men ger en större belastning på länken.
- Om uppdelningen istället sker vid switchen så måste man använda sig av ett protokoll som kan skicka samma data parallellt till flera mottagare, exempelvis Internet group management protocol, IGMP. Nackdelen är att man då inte kan använda sig av TCP eftersom IGMP är ett multicast protokoll och enbart går att implementera över UDP. Se 4.1 för mer information om multicast.

2.4. Uppdelningsproblem

Vid fördelning av data mellan de olika processorkorten i klustret kan man använda olika metoder. Detta avsnitt redogör för hur två vanliga uppdelningsmetoder fungerar samt för och nackdelar med två exempel. Talen i exemplen är påhittade värden. Syftet är att bilda en uppfattning om hur metoderna fungerar samt att ge en grund för vidare implementation.

Parallelliserad uppdelning

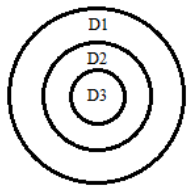
Den totala datamängd som ska beräknas delas upp lika mellan ett antal processorkort i HPC-klustret. Principen för en radar blir då att man delar upp all data som samlats vid ett radarsvep, d.v.s. ett helt varv, så att varje processorkort får en delmängd av hela svepet. Det blir då ett överlapp för varje processorkort. Tänk tillbaks på matrisen i avsnitt 2.2. Det är den radarmatris som delas upp i lika många delar som det finns processorkort. Eftersom matrisen inte går att dela upp exakt kommer en delmängd av varje uppdelad mängd att hamna hos nästa processorkort. Se exemplet i figur 6.



Figur 6. I detta fall ska en 3x3 matris delas upp till 2 st 2x3-matriser. Varje 2x3-matris kommer då att innehålla den dubbelfärgade mängden. Det är den dubbelfärgade mängden som kallas för överlapp.

Om t.ex. överlappet är på 5 %, blir överlappet i datamängd $X * 0.05 * N$. Där X är datastorlek på mängden som skickas och N är antalet processorkort. Med tre processorkort enligt figur 7 hade 10MB data blivit $10 * 0.05 * 3 = 1.5MB$ överlapp.

Denna metod blir mindre effektiv desto fler processorkort man använder. En fördel är dock att det inte blir någon latens på de data som skickas eftersom all data skickas på en gång. Med andra ord går det att beräkna all data på en gång när den har skickats.



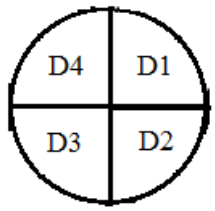
Figur 7. Ett radarsvep uppdelat med parallelliserad uppdelning. D1, D2 och D3 är numret på det processorkort som svepet fördelats till.

Round-Robin uppdelning

Round-Robin används mycket inom schemaläggning och allokerar då en liten bit tid, en så kallad ”time-slice”, för varje process eller uppgift som ska utföras. Dessa processer ligger vanligtvis i en slags kö och processen som är först i kön får en time-slice på sig att utföras för att sedan lämna över till nästa process i kön. Tiden som en process får på sig att utföras får inte vara för kort och inte heller för lång för att uppdelningen ska bli så effektiv som möjligt. Se [15] för mer information om Round-robin.

Med round-robin i en radartillämpning, delar man upp hela svepet i olika delar under svepets gång vilket innebär att beräkningen får en viss latens och även en viss överlappning. Överlappningen blir dock inte lika stor som för parallelliserad uppdelning (t.ex. 0.4% per processorkort). Överlappen blir mindre eftersom storleken på den mängd som delas upp är betydligt mindre än vad den blir för parallelliserad uppdelning.

För att illustrera generellt hur det fungerar kan man förklara med ett enkelt exempel:



Figur 8. Ett radarsvep uppdelat med round-robin. D1, D2, D3 och D4 är numret på det processorkort som svepet fördelats till.

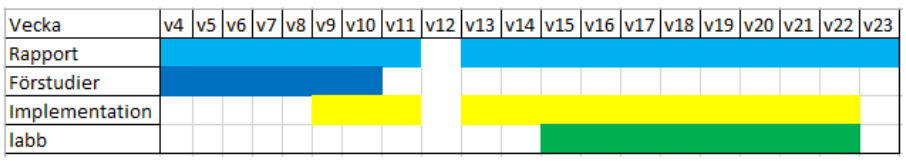
Låt oss säga att vi har fyra processorkort (D1, D2, D3 och D4), samt fyra uppdelningar, att den totala tiden för ett svep/varv är 400ms och varje processorkort har en beräkningstid på 400ms. Det innebär att D1 är färdig med sin beräkning när ny data kommer från nästa svep. Beräkningstiden för ett processorkort blir då antalet noder multiplicerat med beräkningstiden (4*400ms). Detta är ju ett ideellt exempel där ingen hänsyn till latens har tagits och noderna kommer precis att hinna med beräkningen. Latensen för exemplet blir då i värsta fall: 400-400/4 = 300ms, d.v.s. beräkningstiden subtraherat med ”sektortiden”. Sektortiden är den tid det tar för radarsvepet att utföras. Om man tar hänsyn till överlappning i bildexemplet ovan och att ett svep skulle motsvara data på 10MB skulle man få en överlappning på 10*0,004 = 0,04MB, vilket är ganska lite.

3. Metod

Detta avsnitt beskriver på vilket sätt arbetet har gått till och hur det är upplagt.

3.1. Arbetsmetoder

Examensarbetet är indelat i fyra delar: skrivning av rapport, förstudier, labb samt implementation. Stort fokus har lagts på att mäta prestanda mellan två noder d.v.s. implementationen, se diagrammen i avsnitt 6 (Resultat). Själva implementationen är indelad i tre steg: implementation på en dator, mellan två datorer och slutligen implementation på HPC-kluster.



Figur 9. Gantschema för examensarbetet.

Rapport

Rapporten har skrivits parallellt med arbetet, oftast i form av ett dokument som beskriver ett visst moment. Dokumenten användes sedan när rapporten färdigställdes.

Förstudier

I förstudierna har information tagits fram om olika protokoll och samtidigt har protokoll kunnat uteslutas.

Labb

Slutimplementation på företaget där implementationen körs på klustret.

Implementation

Implementationen utgår från en dator, sedan vidare till två datorer och slutligen implementationen på företaget. Implementationen har varit en iterativ arbetsprocess då nya problem ständigt dök upp, oftast till följd av lösningen på ett annat problem.

Arbetet har utförts till största del på egen hand på antingen skolan eller hemifrån och en avstämning med handledare på Saab har skett ungefär en gång i veckan. Dessa möten har varit till för att redovisa det som gjorts samt för att diskutera fram problemlösningar. Ett möte med handledare på skolan har också skett veckovis. Utvecklingsmetodiken under implementationen har varit scrum där en icke-fysisk scrumboard har använts.

3.2. Förstudier

Första uppgiften var att ta reda på vilket protokoll som lämpar sig bäst för problemet. För att kunna utesluta protokoll och för att få en bra övergripande bild som möjligt, samlades information om de vanligaste protokollen in. Poängen med detta var att hitta ett antal intressanta protokoll och göra en litteratursökning på dessa samt att utesluta de protokoll som inte var lämpliga. Eftersom protokollen delas upp i lager var det lämpligast att börja i lagret under tillämpningen, d.v.s. på transportlagret. På denna nivå är TCP och UDP de vanligaste protokollen. Det är därför vanligt att andra protokoll på denna nivå härstammar ifrån antingen TCP eller UDP. Dessa två blev därför en utgångspunkt för arbetet. Se Avgränsningar i 1.4 för mer detaljer kring detta beslut.

Eftersom någon typ av multicast kan vara lämpligt är UDP det protokoll som till en början verkar mest intressant. Det finns en grupp av IP-protokoll, så kallade IP-Multicast där bland annat protokollet IGMP och PIM finns med och de båda använder sig av UDP på transportlagret(se 4.1). Multicast är en funktion som gör det möjligt att skicka data till en grupp mottagare samtidigt, över en länk. Läs mer om multicast och hur det fungerar i 4.1. Det finns även ett annat protokoll på transportlagret som heter SCTP vilket är en slags kombination av TCP och UDP.

Artikel [3] beskriver funktionerna hos SCTP och förklarar även hur SCTP eliminerar en del av de problemen som finns med TCP.

3.3. Testimplementation

För att kunna göra en testimplementation behövdes kunskap om socketprogrammering i java. Den lämpligaste informationen går att hitta på Javas hemsida men även i javadoc, ([11] och [12]).

Den första implementationen var ett program som skickade en fil från en klient till en server som tog emot filen. Detta gjordes för TCP men var mycket svårare att göra med UDP. Eftersom innehållet i paketen inte är av betydelse för mätningens skull övergick programmet istället till att skicka bytevektorer. Med TCP kan man använda sig av en outputstream, för att skriva ut alla byte från en bytevektor eller en fil, till det övre lagrets buffert. En buffert är en slags kö i operativsystemet där paket läggs innan de skickas över en länk eller hämtas ner av ett program. Med UDP lägger man istället in bytevektorn i ett paket och sedan skickar man det. Se [11] för mer information om UDP och TCP i Java.

När detta väl fungerade byggdes koden vidare. Mottagaren och sändaren gjordes om till trådar, bland annat för att flera sändare och mottagare skulle kunna köras samtidigt. Se 5.2.

3.4. Källkritik

Boken Computer Networking tillsammans med RFC (Request for comments) har använts för att hitta information om de olika protokollen. Egentligen är det RFC som har varit mest informativt och sedan har Computer Networking mer varit för att förtydliga vissa delar, eftersom boken är mer pedagogisk i sina beskrivningar. Författarna till boken är kända inom datorkommunikation och boken används även på två kurser i datateknik på Campus Helsingborg. RFC är ett flertal dokument som IETF – Internet Engineering Task Force har publicerat som beskriver internets standard.

Många vetenskapliga artiklar eller böcker som handlar om internet och datorkommunikation, inklusive Computer Networking, hänvisar till dessa dokument och de känns därför som lämpliga referenser.

Eftersom arbetet grundar sig hos radartillämpningar har boken ”Radar target detection – A handbook of theory and practice” använts. Handledaren på företaget har varit till stor hjälp för att förenkla denna information. Handledaren på företaget hänvisade till denna bok själv eftersom examensarbetets grund delvis bygger på de teorier som beskrivs i boken. Källan anses därför tillförlitlig.

Javadoc som bland annat beskriver java.net och java.io har använts som referens till de implementationer som berör UDP och TCP. I vissa fall har inte den informationen som stått i javadoc varit tydlig nog och hjälp har då genom att söka efter problemet på Google. Framförallt har hjälp och inspiration hämtats från ett programmeringsforum som heter Stack Overflow. Anledningen till att det gick att lita på Stack Overflow var att det som skrevs där gick att testa, informationen dömdes därefter tillförlitlig om det fungerade.

4. Teknisk bakgrund

Detta avsnitt förklarar de olika tekniska delarna i arbetet, hur de har använts och det som förstudierna gav som resultat.

4.1. Nätverksprotokoll

Detta avsnitts beskriver en del viktiga egenskaper för de protokoll som undersökts samt vad som skiljer dem åt.

Protokoll	Tillförlitligt	Header	Realtidseg.	CongestionCont.	Handskakning	Tidsgarantier
TCP	Ja	20 bytes	Begr/inga	Ja	3-way	Nej
UDP	Nej	8 bytes	Ja	Nej	Ingen	Ja
SCTP	Ja	12 bytes	Ja	Ja	4-way	ja

Figur 10. En tabell över viktiga egenskaper hos de tre undersökta protokollen på transportlagret.

4.1.1. TCP – Transport Control Protocol

En TCP-anslutning använder en full duplex anslutning vilket innebär att både sändare och mottagare sickar eller tar emot data samtidigt. TCP garanterar att data levereras till applikationen utan fel och i rätt ordning. Efter att varje segment har skickats så startas en timer, har inte sändaren mottagit ett ack på den tiden skickas segmentet igen, därför kan många paketförluster leda till förseningar. ”ack” står för acknowledgement och är ett paket som skickas tillbaks till sändaren från mottagaren för att tala om att ett paket har mottagits. TCP-headern är oftast 20 bytes, vilket är 12 bytes mer än för UDP. Se [1] för mer information.

Slow Start

I TCP finns en funktion som kallas slow-start som är till för att sändaren inte ska skicka mer data än vad mottagaren hinner ta emot. När TCP börjar sända data görs detta med en låg hastighet som ökar exponentiellt. Efter varje RTT (Round Trip Time), dubblas congestion window och den minskar först när en paketförlust sker.

Congestion window size(cwnd) sätts då till 1 och ssthresh (slow start threshold) sätts till MSS/2 (värdet på MSS då paketförlusten skedde). Med MSS avses maximum segment size, vilket är den största tillåtna storleken på ett paket. Slow start börjar sedan på nytt och när cwnd = MSS/2 hoppar TCP över till congestion avoidance mode och då ökas cwnd mer försiktigt för att undvika paketförlust. Congestion är engelska för trängsel, vilket är en vanlig orsak till paketförlust. Se [1] för mer information.

Congestion control

Paketförlust beror oftast på att det bildas köer i en router eller switch vilket kan leda till trängsel och att paket kastas. Congestion control är till för att hindra noder i ett nätverk att orsaka trängsel. Detta begränsar oftast bandbredden vilket kan ha en negativ effekt på video- och ljud-streamingtjänster som har minimala genomströmningskrav, d.v.s. bandbredden är viktigare än paketförlusten. Man har därför sett att många utvecklare väljer UDP istället för TCP till realtidsapplikationer enligt [1].

4.1.2. UDP – User Datagram Protocol

UDP är lite svårare att använda än TCP, eftersom mer måste göras på applikationsnivå av programmeraren. Risken för paketförlust är vid denna tillämpning inte särskilt stor vilket gör UDP till ett lämpligt protokoll. Orsaken till paketförlust beror oftast på att det bildas köer i routers vilket leder till att paket blir för gamla och slängs enligt [1]. Därför borde risken för paketförlust öka, desto fler routrar och switchar som paketen passerar på vägen till mottagaren. UDP använder ingen handskakning eller synkronisering mellan mottagare och sändare. Det går därför inte att garantera att paketen kommer fram eller att de är i rätt ordning när de kommit fram. UDP-headern har oftast en storlek på 8 bytes. UDP kan skicka data med den hastighet som önskas, men genomströmningen kan bli mindre pga. mellanliggande länkar med en lägre genomströmningskapacitet eller trängsel(congestion). Mer om UDP går att läsa om i [1].

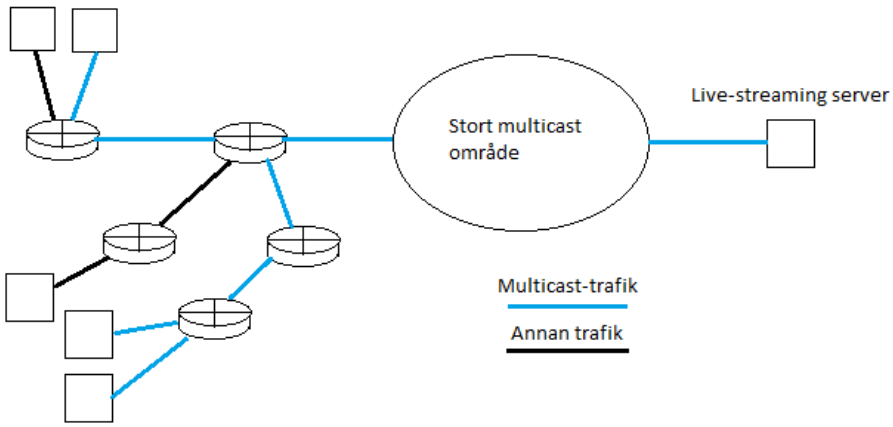
IGMP – Internet Group Management Protocol

IGMP är ett protokoll på nätverkslagret som använder sig av UDP på transportlagret och skickar data som IP-paket.

Protokollet används mellan klienten och routern som är närmst klienten. Det krävs därför ett annat protokoll för att koordinera all multicast trafik och den direkt anslutna routern, så att multicastad data når sin slutdestination. Till exempel kan man använda protokollet PIM (Protocol Independent Multicast) för koordination.

Klienter som är anslutna i samma multicastgrupp tar emot data från t.ex. en och samma källa. Exempelvis en livesändning där samma data skickas till ett x antal klienter som använder en streamingtjänst enligt [1].

Uppsättningen på gruppen blir antingen som ett ”grupp-baserat träd” eller ett ”käll-baserat träd”. Antingen distribueras trafiken för alla sändare med samma träd(grupp-baserat) eller distribueras trafiken specifikt för varje sändare med ett träd för varje sändare(käll-baserat). Anledningen till att träd byggs är för att minska onödig trafik genom att försöka hitta en väg där trafik enbart går via de routrar som är anslutna till källor i multicast-gruppen. IGMP gör det möjligt för en värd att säga till sin direkt anslutna router att en applikation som körs på värden vill gå med i en viss multicast-grupp. Eftersom IGMP enbart existerar mellan värd och närliggande router, behövs ett annat protokoll för att koordinera routerna [1]. Se figur 12 samt [1] för mer information om multicast.



Figur 11. Multicast-trafik, till exempel skickas videodata från en livesändning till ett nätverk, ett träd byggs då utefter de källor som är anslutna till streamingtjänsten (multicast-gruppen). De noder som är har en blå linje i bilden är anslutna till samma multicast-grupp.

PIM – Protocol Independent Multicast

PIM är ett protokoll som är till för att hjälpa multicast-routerna att koordinera trafik till olika destinationer. Protokollet har ett antal olika sätt att göra detta på:

PIM sparse mode: Bygger enkelriktade delade träd som börjar vid en viss samlingspunkt, ”rendezvous point”, per grupp. Se [5] för mer information.

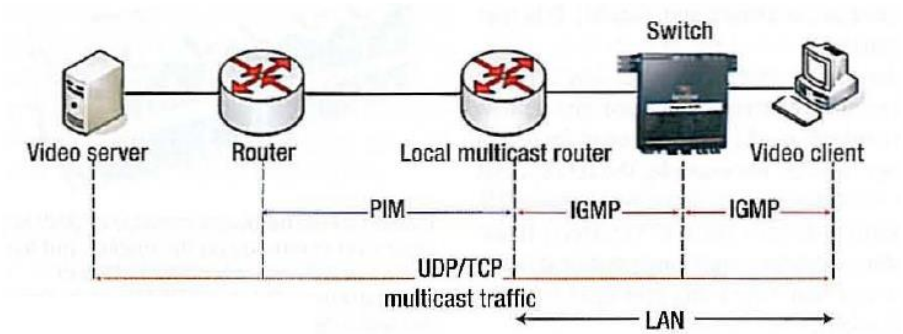
PIM dense mode: Bygger ett träd genom att skicka data över en hel domän. De grenarna i trädet som inte har en mottagare skärs sedan bort. Se [6] för mer information.

Bidirectional PIM: bygger delade tvåvägs träd men de är uppbyggda för att hitta kortaste vägen, vilket ger en längre end-to-end delay än PIM sparse mode. Se [7] för mer information.

PIM source-specific multicast: Bygger träd som grundar sig i enbart en källa vilket ger en mer säker och skalbar modell för ett begränsat antal applikationer som oftast lämpar sig för någon typ av sändning.

I SSM (source-specific mode) skickas ett IP-datagram till en källa ”S” och en SSM destination ”G”. De klienter som ska motta datagrammet gör det genom att abonnera till kanal (S, G). Mer detaljer går att läsa i [8].

Alla dessa varianter kräver också ett annat protokoll för att överföra data från slutrouter till slutdestination, till exempel IGMP eller ICMP. (Se figur 12).



Figur 12. Illustrerar ett exempel på multicast i ett nätverk [30].

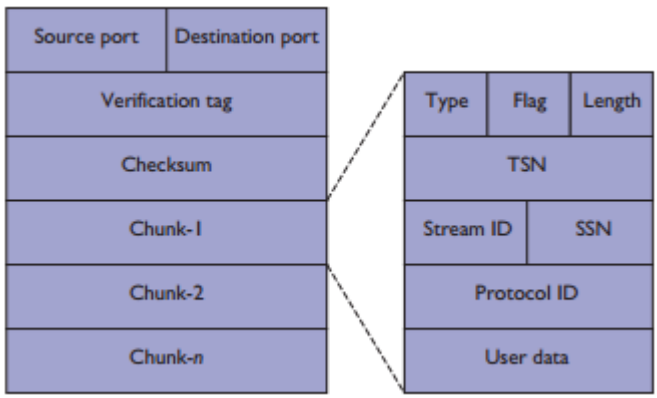
RTP – Real-Time Transport Protocol

RTP är specifikt framtaget för att transportera audio och video och körs ovanpå UDP. Sändaren kapslar in ett stycke media i ett RTP-paket som sedan kapslas in i ett UDP segment enligt [1]. Headern är normalt 12 bytes stor och innehåller ett sekvensnummerfält som gör det möjligt för klienten eller applikationen att upptäcka paketförlust. Se [1] för mer information.

4.1.3. SCTP – Stream Controlled Transport Protocol

SCTP är ett nytt protokoll i transportlagret och är precis som TCP ett protokoll som tillhandhåller pålitlig dataöverföring. Protokollet introducerades i oktober, år 2000. Vidare är SCTP ett protokoll som kräver att en anslutning upprätthålls mellan de två noderna som data skickas mellan. En av de största fördelarna hos SCTP är dess multi-streaming funktion. Data delas då upp i flera olika strömmar där varje ström har en självständig ordning som data levereras i. Sker det en paketförlust i någon av strömmarna påverkas inte de andra. Till skillnad från TCP där en paketförlust orsakar en försening tills data skickas i rätt ordning igen.

Multihoming är en annan användbar funktion hos SCTP, där alternativa adresser används ifall en nod i ett nätverk fallerar. Detta kan i normala fall innebära paketförluster och omsändningar som inte når fram under tiden noden återhämtar sig. Detta kallas med ett annat ord för rekonvergens. Med hjälp av de alternativa adresserna kan istället SCTP återsända paket en alternativ väg. Dock hjälper det inte mycket ifall det bara finns en väg och det blir ett så kallat ”Single point of failure”. Se [9] och [32].



Figur 13. En typisk header för SCTP.[9]

Det som är mest intressant i headern är ”Verification tag”. Denna innehåller ett värde som utväxlats under handskakningen när SCTP-förbindelsen upprättades. Om ett SCTP-paket inte innehåller denna tag slängs paketet direkt hos mottagaren. Verifiacation tag skyddar mot gamla paket från en gammal association eller från man-in-the-middle attacker. Man-in-the-middle attacker innebär när t.ex. två program, A och B, kommunicerar med varandra och en tredje okänd part lyssnar av eller manipulerar det som skickas mellan A och B. Det eliminerar även problemet hos timed-wait state hos TCP. Timed-wait state används i TCP när en anslutning upphör och ett sista ack skickats. Servern eller klienten som skickade acket väntar då en viss tid innan en ny anslutning kan göras, för att inte riskera överlappning. Se [9].

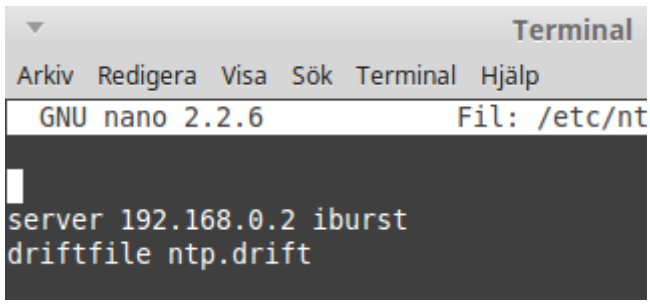
SCTP används mest inom telecom och VoIP(voice over IP) och finns implementerat i operativsystemen, Linux, BSD och som tredje-part i Windows. Både FTP och SIP (Session Initiation Protocol) bygger på funktioner hos SCTP. Kommersiellt finns även protokollet tillgängligt i Ciscos IOS (Internetwork Operating System), som är ett operativsystem för deras routrar och switchar. Se [31].

4.2. Linux

För att göra implementationen på två datorer användes två stationära datorer som Linux Mint version 17 installerats på. Dessa kopplades sedan ihop med hjälp av vars en ethernetkabel och en 1-Gbits switch. Datorerna hade dessutom samma sorts nätverkskort men en kapacitet på 1Gbit/s. Anledningen till att Linux valdes som operativsystem var för att slutimplementationen på Saab sker i samma miljö. Om man inte har använt Linux tidigare kan det vara bra att lära sig hur man använder terminalfönstret och dess grundläggande kommandon. Linux är betydligt mycket lättare att använda om man vet hur man använder terminalfönstret. Terminalfönstret i Linux påminner lite om kommandotolken i windows, dock fyller den en mycket större funktion, då många program körs direkt i terminalfönstret. Fördelen med Linux är att det finns en hel del användbara program som är till stor hjälp när man sysslar med utveckling, till exempel netcat och netstat för att nämna några. Även NTP följer med de flesta Linux versioner men det följer inte med Mint 17, utan måste installeras på egen hand. Netcat är till för att läsa eller skriva till olika nätverksanslutningar och kan till exempel användas för att skicka filer över nätverk med UDP eller TCP [17]. Netstat används bland annat för att man ska kunna få information om olika nätverksanslutningar. Till exempel kan man se vilka portar på datorn som används, vilka protokoll som använder portarna och dessutom hur routing tabellerna ser ut för anslutningarna, för att nämna några.

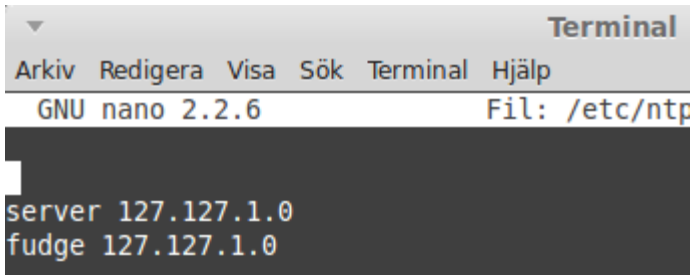
4.3. NTP – Network Time Protocol

NTP är ett protokoll som är till för att synkronisera systemklockan hos en dator men en annan dator eller server. Protokollet synkroniserar med en noggrannhet på mindre än en millisekund på LAN och upp till ett fåtal millisekunder på WAN. För denna implementation användes NTP version 4 (se [10]) som går att hämta gratis på www.ntp.org. Syftet var att låta datorn som skickar paket synkronisera mot datorn som tar emot paket, eftersom det är hos mottagaren som tidsuträkningen kommer att ske. Innan man kan använda NTP måste man skapa en konfigurationsfil så att protokollet vet vilken server den ska synkronisera mot. Så här såg det ut i sändarens konfigurationsfil:



Figur 14. Screenshot från konfigurationsfilen hos datorn vars tid synkroniseras. Server 192.168.0.2 är den adress klockan ska ställas in efter. Iburst reducerar en inledande försening som annars förekommer när NTP ska börja ställa in klockan. Driftfilen är en inställningsfil som innehåller ett värde som NTP använder för att justera tiden.

Så här ser mottagarens konfigurationsfil ut:



Figur 15. Även datorn som ska agera server måste ha NTP igång. NTP hämtar tiden från serveradressen, vilket är datorns egen adress. Fudge innebär att NTP ska hämta tiden lokalt från en adress, ifall det inte går att hämta från server-adressen.

För att sedan undersöka om datorerna är synkroniserade kan man testa att ändra sändar-datorns klocka. Genom att skriva ”ntpq –p” i terminalfönstret kan man då se att parametern ”offset”, d.v.s. tidsskillnaden mellan de båda datorerna, var ett högt antal millisekunder. Efter ett antal minuter ställdes sändardatorns klocka till samma tid som hos mottagaren. En tredje sak man kan kolla är om driftfilens värde har ändrats för då vet man även att protokollet försöker synkronisera tiden. Protokollet använder värdet i driftfilen som startvärde, så att den slipper att synkronisera om datorn från början varje gång man stänger av och sätter på NTP. Värdet är representerat i PPM (parts per million) där 1 PPM = 1 microsekund/sekund = 3.6 ms/timme = 86.4ms/dag osv.

Trots att NTP fungerade som det ska så var inte dess synkronisering tillräckligt noggrann för att tidmätningen skulle bli rätt. Detta beror på tiden förskjuts mellan de båda datorerna hela tiden eftersom deras interna klockor går med olika hastighet. Tidsskillnaden ändras då hela tiden och det är bara med ett visst intervall som tiden kan ställas in med NTP för att schemalagda aktiviteter i datorn inte ska bli störda. Detta innebär en tidsförskjutning som inte är konstant vilket innebär att varje mätning kommer innehålla tiden det tog att skicka paketet tillsammans med tidsförskjutningen [13].

4.4. Socketprogrammering i Java

För att ett program ska kunna kommunicera med ett annat program över ett nätverk måste de meddelanden som skickas transporteras via de undre nätverkslagren. Ett program använder sig då av ett gränssnitt som kallas för socket för att både ta emot meddelande och skicka meddelanden. Det är alltså en socket som ser till att operativsystemet och de nätverkslagren som ligger under applikationslagret skickar eller tar emot ett meddelande. Detta innebär att man som programmerare har full kontroll över det som händer på applikationsnivå, men väldigt lite kontroll över det som händer på de andra nivåerna. Se [1] för mer detaljer.

I javas socket finns det en del metoder som gör att man kan justera vissa inställningar på transportlagret, men det finns inte mycket man kan ändra på. Som exempel kan man i TCP:s socket med hjälp av TCP_NoDelay stänga av Nagle's algoritm. Nagle's algoritm är till för att paket inte är för små när de skickas iväg vilket kan ge en viss fördröjning för tidskritisk data men eliminerar att mottagaren får en onödig overhead. Se [18] för mer detaljer kring Nagle´s algoritm.

Med UDP:s socket behöver man inte nödvändigtvis initiera en mottagar- eller sändarsocket med en specifik adress och portnummer utan kan göra det under körning med hjälp av socket.bind() metoden. Detta kan vara användbart om man vid programmets start inte vet vem man ska skicka data till. Se [11] för mer information.

Det går alltså att ändra vissa inställningar men man kan inte göra stora ändringar på protokollet, väljer man UDP får man också förvänta sig ett mer opålitligt protokoll och väljer man TCP får man förvänta sig ett betydligt långsammare men samtidigt ett mer pålitligt protokoll.

5. Analys och implementation

Detta avsnitt beskriver hur arbetet har gått till i detalj. Analysen förklarar varför vissa beslut har tagits och hur slutligen ett resultat uppnåtts. Själva koden som arbetet har nått fram till finns i Appendix A och resultatet finns i avsnitt 6.

5.1. Utvärdering och analys av protokoll

Vid förstudierna undersöktes ett antal andra protokoll, än TCP och UDP, för att undersöka ifall det finns alternativ till dessa. Flertalet av protokollen som undersöktes var på lager ovanför transportlagret, dels för att det inte finns så många protokoll på transportlagret men även för att se om det finns protokoll som i kombination med något av protokollen på transportlagret hade varit ett bra alternativ. Anledningen till att protokoll under transportlagret inte undersökts är för att data i HPC-klustret skickas mellan två program, vilket sköts av transportlagret, vilka protokoll som används under dessa är redan förutbestämt. Det som skrivs här baseras på fakta från 4.1 Nätverksprotokoll.

TCP och UDP

För att TCP ska fungera i det här fallet måste en enskild förbindelse upprättas från radarn till varje processorkort. Switchen måste fördela paketen mellan varje processorkort och alla ack-paket som skickas tillbaks. Denna lösning skulle kräva flera TCP-förbindelser, en för varje processorkort i HPC-klustret. Många förbindelser kommer dock att orsaka en större belastning på den fysiska länken mellan radarn och processorkorten vilket innebär att bandbredden för varje förbindelse sänks. Belastningen orsakas i detta fall av att det blir mycket overhead. För UDP kommer det istället att innebära paketförlust. Anledningen till att flera förbindelser måste göras med TCP är att för att en TCP-förbindelse enbart kan existera mellan två noder. Med UDP kan man antingen skicka data mellan flera förbindelser eller skickar man all data på en gång som delas upp vid switchen med hjälp av multicast. Se 4.1.

De båda protokollen har olika för- och nackdelar, att ställa dem mot varandra är därför det lämpligaste sättet för att ta reda vilket av de två protokollen som är bäst.

En teoretisk beräkning kan vara bra om man vill veta hur protokollen skiljer sig åt på transportlagernivå, men går inte att använda för att se skillnaden på applikationsnivå. Detta beror på att det är allt för många faktorer som inte går att ha med i en beräkning, till exempel fördröjningen som sker när Javas instruktioner schemaläggs i operativsystemet. Att använda de båda protokollen i en implementation för att ta reda på vilka begränsningar som finns hos de båda på applikationsnivå är därför lämpligt. För mer information om protokollen se 4.1.

Multicast (IGMP & PIM)

Eftersom data ska skickas från en källa till flera processorkort är multicast-protokollen lämpliga att undersöka. I 4.1 nämns det att multicast enbart går att tillämpa ovanför UDP, vilket innebär att man först bör kunna utesluta TCP. Om UDP visar sig vara ett bättre alternativ än TCP kan det vara lämpligt att överväga multicast-protokollen. Det kan då vara intressant att testa IGMP för att se om resultatet blir bättre än vad det blir för en implementation med flera förbindelser över en länk. För mer information om protokollen se 4.1.2.

RTP

Detta protokoll ligger på applikationsnivå ovanför UDP. Det lägger till en header på paketen som bland annat innehåller ett sekvensnummer för att man ska veta i vilken ordning mottagna paket har skickats. Protokollet används mycket inom realtidsapplikationer för audio och video vilket gör det lämpligt att använda inom radartillämpningar. Detta är också ett alternativ som är värt att överväga ifall UDP visar sig vara det bästa alternativet.

SCTP

Anledningen till att detta protokoll verkade intressant var att det är designat för realtidsapplikationer, men bibehåller ändå de egenskaper som finns hos TCP.

Det finns implementerat i javas bibliotek och är tillgängligt för allmänheten, det är dock inte implementerat i javas standardbibliotek (java Standard Enviroment). Anledningen till detta är att industrin anses ha för lite erfarenhet av protokollet enligt [32]. Vid informationssökning var det svårt att hitta bra information som beskriver protokollets egenskaper och ännu svårare att hitta konkreta exempel på hur man använder sig av SCTP i Java. Det bedömdes därför inte aktuellt att använda SCTP i en implementation på grund av bristfällig information. För mer fakta om SCTP se 4.1.3.

5.2. Implementation

För att genom implementation ta reda på vilket av UDP och TCP som är mest lämpligt finns det två viktiga egenskaper att jämföra. Hur snabbt kan de båda protokollen skicka data innan, för applikationen, oacceptabla paketförluster inträffar och hur latensen påverkas. Detta går inte att få fram teoretiskt utan måste testas i praktiken. Först genom att testa på en dator, sedan mellan två datorer och slutligen på klustret. För att testa detta designades programmen så att man kunde mäta mottagna paket och dess fördröjning. Alla uppmätta resultat har gjorts med en noggrannhet på 50 körningar, varefter ett medelvärde tagits fram som överensstämmer med uppmätta värden från varje enskild körning.

5.2.1. Implementation på en dator

Den första följden av testimplementationer var enbart för att förstå hur socketprogrammering fungerar. Ett program som skickade ett paket med en given storlek gjordes då för både UDP och TCP. Tanken var att implementationen skulle visa om det var möjligt att komma upp i en hög hastighet genom att skicka och ta emot data lokalt, alltså var inget nätverk inblandat. Se appendix A.

Nästa steg var att se hur prestandan blir om man delar upp programmet i olika processer, en process för att skicka data och en för att ta emot data. Principen är samma fast här uppkom först ett nytt problem som gäller de båda protokollen. Tidmätningen kan ju inte längre göras på samma sätt där eftersom programmen nu är helt skilda.

Detta kunde lösas genom att man på ett lämpligt sätt skickar med tiden från sändartråden. För att skicka med tiden i paketet är man tvungen att använda sig av en Bytebuffer för att lägga in värden i bytevektorn som man sedan skickar. Se appendix A.

Sista steget var att låta sändar-klassen implementera Javas TimerTask och sedan införa en timer som anropar denna med ett fixt intervall. Detta för att man ska kunna skicka en visst bestämd mängd data per sekund. Se appendix A.

UDP

I den första implementationen som gjordes med UDP var målet att enbart skicka data mellan två sockets.

```
public void run() {
    try {
        // eftersom paketet ska skickas till samma dator kan man hämta den
        // lokala addressen.
        InetAddress adrs = InetAddress.getLocalHost();
        socket = new DatagramSocket();
        while (running) {
            // tidsmätning
            currentTime = System.nanoTime();
            // createPacket skapar en byte-vektor med storlek 1500 bytes
            sendByte = new byte[1500];
            // här skapas själva datagrampaketet som innehåller vektorn, IP-adressen och portnummer
            packet = new DatagramPacket(sendByte, sendByte.length, adrs, 6070);
            // paketet skickas sedan med hjälp av send till porten direkt
            socket.send(packet);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figur 16. En metod som skapar och skickar paket med UDP i Java.

Det kanske känns meningslöst att skicka tomma bytevektorer, men de tar lika mycket plats som om de hade innehållit någon typ av data. Om man istället vill skicka filer kan man använda FileInputStream, som är en klass i java, för att överföra filen till en bytevektor.

I mottagarklassen tas data emot ungefär på samma sätt:

```
public void run() {
    try {
        // skapar socketen som ska motta data med porten den tar emot data på
        socket = new DatagramSocket(6070);
        // skapar bufferten som ska lagra de mottagna bytesen
        sentBytes = new byte[1500];
        // skapar paketet som data ska lagras i
        packet = new DatagramPacket(sentBytes, sentBytes.length);
        // lyssnar efter paket och hämtar upp paketet till applikationsnivå,
        // när paketet är mottaget kommer packet att innehålla de skickade bytesen
        socket.receive(packet);
        // gör något med den mottagna datan...
```

Figur 17. En metod som tar emot paket i UDP i Java.

I nästa implementation behövdes en metod för att skapa paketen som ska skickas och för att lägga in tiden då paketet skickades.

Eftersom mottagaren inte vet när den ska böra lyssna efter paket måste den implementeras så att det kan ta emot paket utan att veta när paketen kommer. Metoden som skapar paket kan lätt utvidgas så att den även skickar med ett sekvensnummer, på det sättet kan mottagaren få reda på om det skett en paketförlust. Därför gjordes två implementationer, en som bara skickade data tills den var klar och en som skickade data och väntade på en acknowledgement mellan varje paket, ungefär som Stop-And-Wait.

Den sista implementationen gjordes för att förbereda inför den slutliga testen mellan två datorer. Tanken var att programmet med hjälp av en Timertask skulle skicka en viss mängd data med ett intervall på en sekund.

```
SendFile sf = new SendFile(1000000);
Timer timer = new Timer();
timer.scheduleAtFixedRate(sf, 0, 1000);
```

Figur 18. Så här kan TimerTask användas i Java.

Timer.scheduleAtFixedRate() kör run-metoden i tråden i ett intervall på 1000ms = 1 sekund. Om en körning tar för lång tid kommer metoden att köra run-metoden en extra gång för att hinna ikapp.

Något som upptäcktes vid körning var dock att denna metod inte avbryter en while-loop, vilket resulterar i att tiden överskrids ifall det tar längre än en sekund att skicka mängden. Därför behövdes även en kontroll för att se till att while-loopen slutar köras efter en sekund.

TCP

Precis som med UDP så var målet med TCP att enbart skicka data mellan två sockets i en och samma klass. För att skicka data i TCP måste man använda en OutputStream som skriver ut data till TCP:s buffert där TCP själv sköter paketindelning och sändning. Så här ser en enkel metod ut för att skicka data med hjälp av en outputStream i java:

```
public void run(){
    try {
        // skapar en socket med mottagarens portnummer och
        // en bufferedOutputStream för att skicka data.
        sendSocket = new Socket("", 5555);
        out = new BufferedOutputStream(sendSocket.getOutputStream());

        // sålänge all data inte har skickats
        while (curDataSize > 0) {

            // skriv ut data till outputStream
            out.write(dataPacket);
            // tvingar outputstream att skriva ut all data till TCPs buffert
            out.flush();
```

Figur 19. En metod som skickar data med TCP i Java.

Så här ser en enkel mottagare ut:

```
public void run() {
    try {

        // Definera tcp-socket
        srvSocket = new ServerSocket(5555);
        // Accepterar anslutning från sändare
        socket = srvSocket.accept();
        // Timeout, stänger socket om ingen data mottagits på 10s
        socket.setSoTimeout(10000);
        int count = 0;
        // initierar en ny bytevektor att lagra ett "paket" i.
        data = new byte[pktSize];

        // lokal inputstream som hämtar data från kön
        is = socket.getInputStream();

        // sålänge data läses från kön ska datan som läses behandlas
        while ((count = is.read(data)) > 0) {
            // gör något med mottagen data
```

Figur 20. En metod som tar emot data med TCP i java.

I den andra implementationen behövdes även här en metod för att skapa paket. På samma sätt som med UDP läggs tiden paketet skickats i paketet, så att man ska kunna beräkna tiden det tar att skicka.

5.2.2. Implementation på två datorer

Nästa steg var att implementera de båda testprogrammen så att data skickas mellan två datorer och sedan göra en mätning. Det som är intressant att mäta är hur paketförlusten påverkar UDP och TCP samt till vilken gräns data kan tas emot beroende på hur mycket man skickar. Programmen behövde inte ändras särskilt mycket för att de skulle fungera på två datorer, problemet var istället att det dök upp ett antal nya problem som skulle lösas. När allt väl fungerar som det ska är tanken att mätningen ska resultera i sex olika grafer, tre för varje protokoll.

Försök 1

De programmen som skrivits behövde modifieras så att sändaren skickade till rätt IP-adress och inte den lokala adressen som programmet gjorde tidigare. När detta väl var ordnat gick det att skicka data mellan datorerna, men då uppstod ett annat problem.

För både TCP och UDP skickas ju tiden med i paketet och sedan används tiden hos mottagaren för att beräkna tiden det tar att skicka ett paket.

I programmen används metoden System.NanoSeconds() för att hämta nuvarande tid och enligt [11] så är det run-time för den JVM som kör programmet. Men tiden blev inte rätt vilket kunde bero på att datorernas klocka inte är samma. För att mäta tiden kan man göra på två sätt, antigen mäter man genomsnittliga RTT/2 för alla paket eller synkroniserar man tiden för de båda datorerna. Eftersom det för tidmätningen är intressant att se vad som händer med tiden det tar att skicka ett paket när protokollet blir hårt belastat, kommer inte det första alternativet att fungera. Det beror på att sändaren måste vänta på samma paket från mottagaren innan den kan skicka igen, de båda protokollen kan då inte skicka data lika snabbt.

Nästa steg var att försöka synkronisera tiden på något sätt. Ett vanligt sätt att synkronisera tiden hos datorer på exempelvis ett nätverk är att använda sig av ett protokoll som heter NTP.

Försök 2

Vid nästa försök användes NTP för att synkronisera datorerna (se 4.3). Poängen med detta försök var att se hur mycket data man kan skicka på en sekund innan paketförlusten blir för stor. Trots att NTP användes fanns tidsskillnaden fortfarande där men istället uppkom ett nytt problem som inte togs i åtanke vid förra försöket. När data skickas mellan datorerna med UDP utan stop-and-wait, hinner inte mottagartråden att plocka ner data från mottagarbufferten tillräckligt snabbt. Stop-and-wait innebär att ett ack skickas tillbaka till sändaren efter varje mottaget paket. För TCP fick istället data slängas hos sändaren för att det tog mer än en sekund att skicka. Se figur 21.

Skickad mängd	Mottagen mängd(UDP)	Mottagen mängd(TCP)	Förlorad mängd(UDP)	Förlorad mängd(TCP)
10000	10000	10000	-	-
100000	100000	100000	-	-
1000000	1000000	1000000	-	-
10000000	10000000	10000000	-	-
20000000	20000000	20000000	-	-
30000000	22876500	30000000	7123500	-
100000000	23329500	18567000	76670500	879086500

*Figur 21. Mängden representeras i bytes. *Efter att 30 000 000 bytes hade skickats ett antal gånger hann inte TCP med att skicka all data. Detta kan bero på att congestion control fick sänka bandbredden för att mottagaren skulle hinna plocka ner data från bufferten.*

För att se om detta var ett problem i programmet eller mellan datorerna testades samma program lokalt på en dator. Det visade sig då att samma problem uppkom även här. När buffertstorleken ändrades hos mottagaren fungerade det och alla paket kom fram. Nästa steg var att testa det på de två linuxdatorerna. Det fungerade dock inte att göra samma sak på Linuxdatorerna, paketförlusten blev fortfarande samma. Med hjälp av Netcat i Linux går det att lyssna på mottagardatorns UDP-portar. Man kunde då se att så fort data skickas, fylls mottagardatorns buffert till max direkt och blir till 0 så fort man slutar skicka data. Att dela upp överföringen på länken genom att skicka och ta emot med flera trådar gav samma resultat, eftersom mottagarbufferten ändå förblir samma. Det visade sig att man både behöver sätta buffertstorleken i java och i operativsystemet för att det ska fungera. Enligt kö-teori kommer mottagarens buffert att växa även om intensiteten som paket ankommer med är samma som den intensitet de behandlas med, enligt [1]. Tabellen i figur

22 visar hur betydelsefull buffertstorlekarna faktiskt är.

Problemet gick att lösa genom att man ändrade de maximala buffertstorlekarna i operativsystemet för de båda datorerna. Det räcker alltså inte att ändra dem enbart i java på applikationsnivå utan man måste se till att operativsystemet faktiskt tillåter detta.

För att se hur buffertstorleken påverkar de olika protokollen gjordes ett antal olika mätningar med programmen som redovisas i figur 22.

Paketstorlek	Sändarbuffert	Mottagarbuffert	Mbits/s	Skickad data	Mottagen data
1500	1500	1500	130	1000	873
1500	64000	1500	270	1000	500
64000	64000	163840	800	1000	972
64000	1500	163840	130	1000	986
64000	163840	163840	920	1000	962

Figur 22. Tabell för UDP, där skickad och mottagen mängd representeras i megabytes och paketstorlek samt buffertstorlek i bytes. Oavsett vilken buffertstorlek som sattes gick den inte över 163840 på linuxdatorerna.

Paketstorlek	Sändarbuffert	Mottagarbuffert	Mbits/s	Skickad data	Mottagen data
1500	1500	1500	80	1000	1000
1500	64000	1500	80	1000	1000
64000	64000	163840	300	1000	1000
64000	1500	163840	200	1000	1000
64000	163840	163840	300	1000	1000

Figur 23. Tabell för TCP, där skickad och mottagen mängd representeras i megabytes och paketstorlek samt buffertstorlek i bytes.

Det som är märkbart är att om sändaren och mottagarens bufferts är lika stora, blir bandbredden större. Den stora paketförlusten hos UDP berodde alltså till största del på att mottagaren inte hinner plocka ner paketen i tid så att bufferten fylls. Mätresultaten för UDP visade att när mottagarbufferten var mindre än sändarbufferten tog bara mottagaren emot ungefär hälften av alla skickade paket. Paketförlusten blir i genomsnitt 4 % för UDP om sändarens och mottagarens buffert är tillräckligt stora. Denna procentenhet varierar givetvis beroende på hårdvara och bakgrundsprocesser. En av orsakerna till att det var värt att försöka ändra buffertstorleken i både programmen och i operativsystemet var det resultat som gavs när man istället använde Netcat.

Netcat tog alltid emot ungefär hälften av den fil man skickar när man använder UDP men när istället filen togs emot med mottagarprogrammet (i java) lästes enbart 20 % av bytsen. Ändå överförde programen och netcat data med samma hastighet enligt nätverksstatistiken i operativsystemet. När buffertstorleken ändrats i båda java och i operativsystemen så fungerade det bättre men paketförlusten var fortfarande hög. Genom att undersöka vad detta kunde bero på visade det sig att problem relaterade till hög paketförlust inte nödvändigtvis behöver bero på att paket försvinner utan kan också ha att göra med hur man läser mottagna bytes i Java.

```
while (true) {
    //skapar ett nytt mottagarpaket
    packet = new DatagramPacket(buffer, buffer.length);
    // hämtar paketet från kön
    socket.receive(packet);
```

Figur 24. Visar hur ett paket skapas varje gång innan det hämtas.

Genom att anropa metoden packet.getLength() kan man hämta antalet byte som finns i paketet. För att detta ska fungera korrekt måste ett nytt paket skapas även hos mottagaren varje gång ett nytt paket mottags, annars kommer packet.getLength() enbart skriva ut det minsta antalet mottagna bytes. Genom att ändra detta och buffertstorleken blev paketförlusten betydligt mindre, dock var paketförlusten fortfarande något för stor. Samtidigt är UDP ett protokoll där man bör förvänta sig paketförlust.

Försök 3

Det saknades fortfarande en lösning på hur man skulle mäta tiden det tar att skicka paket utan att använda stop-and-wait. Genom att ta tiden då ett paket skickas subtraherat med tiden ett paket blir mottaget, får man ju tiden det tar att skicka ett paket. Detta fungerar så länge tiden tas från samma källa, d.v.s. tiden mäts från samma klocka. När man gör detta på två datorer är deras klocka inte synkroniserad. Som tidigare nämnt användes NTP för att försöka synkronisera datorerna utan framgång. Det andra alternativet var att mäta Round-Trip-Time, RTT för paketen och sedan dela denna med två, vilket rimligtvis är tiden det tar att skicka ett paket.

Problemet är att mottagaren då måste skicka ett paket med samma storlek tillbaks till sändaren för varje mottaget paket, detta påverkar direkt prestandan eftersom paketen inte kan skickas kontinuerligt utan istället måsta sändaren vänta på ett ack efter varje paket.

Men vad händer om man i början av mätningen istället tar reda på tiden som skiljer mellan de båda datorerna och sedan räknar ut tiden det tar att skicka ett paket med hänsyn till detta? För de båda protokollen kan det då tänkas gå till på följande sätt:

1. Skicka ett initialt paket med UDP som innehåller sändarens nuvarande tidsstämpel till mottagaren och ta sedan mottagarens starttid subtraherat med sändarens starttid. Gör detta ett flertal gånger och räkna ut den genomsnittliga tiden. Nu har man differensen.
2. Skicka sedan de vanliga paketen och ta tiden paketen mottogs subtraherat med tiden paketet skickades subtraherat med differensen och lägg till i den totala tiden.
3. När alla paket mottagits tar man den totala tiden dividerat med antalet mottagna paket och då får man medelvärdeslatensen för varje paket.

Vid implementering på lokal dator visade det sig att detta inte funkade. Problemet var att differensen visar sig vara den tiden som skiljer mellan programmen när de startade. Differensen blev alltså 5-10 sekunder och eftersom tiden det tar att skicka ett paket är betydligt mindre blev varje tidmätning ett stort negativ värde. Tiden som man vill ha reda på är tiden som skiljer mellan de olika datorernas klocka, inte tiden som skiljer mellan när programmen startat.

Försök 4

Istället för att försöka återuppfinna hjulet bör det vara ett så pass generaliserat problem så att det faktiskt finns ett sätt att göra detta på. På forumet Stack Overflow var det en person som hade ett liknande problem och han undrade just hur man mäter tidsskillnaden mellan två nätverksenheter.

Eftersom algoritmen verkade rimlig var den värd att testa. Så här fungerar algoritmen: (Server och klient har bytt plats för att det ska passa programmet, jämfört med hur algoritmen beskrivs på sidan).

c = klientens klocktid. s = serverns klocktid, d = differensen, eller tiden som läggs till hos servern för att tiden ska bli rätt och t = c eller s beroende på var tiden mäts. Detta ger: $d = c - s$

Så här går det till:

1. Skicka ett paket med klientens tid till servern.
2. Servern lagrar tidsskillnaden mellan sin egen tagna tid och tidsstämpeln i paketet som t1.
3. Servern skickar sedan sin egen tagna tid till klienten.
4. Klienten skickar sedan skillnaden mellan tidsstämpeln från servern och sin egen klocka till servern som t2.
5. Servern räknar sedan ut differensen genom att ta: $d = (t2 - t1)/2$

Sambandet ges från:

$t1 = t - d$
$t2 = t + d$
$t1 + d = t2 - d$
$d = (t2 - t1)/2$

Detta görs ett antal gånger för att d ska bli ett genomsnittlig värde.

Notera att t1 och t2 båda innehåller travel-time (tiden för ett paket att färdas från en punkt till en annan i ett nätverk) för paketen + differensen. Om man antar att travel-time är samma i båda riktningar, vilket man kan i detta fall, går det att försumma travel-time.

När detta var implementerat visade det sig att differensen blev vad man kan förvänta sig d.v.s. väldigt liten när mätningen görs på en dator. (ca 2000ns). Tidmätningen som följde blev också mer realistisk (mindre än 0.1ms). Nästa steg var att testa på två datorer.

Försök 5

Att försöka mäta tiden genom att använda differensen fungerade inte. Detta beror till största del på att tidsskillnaden inte enbart beror på tiden som skiljer mellan datorernas klocka utan på att programmen körs i två olika JVM(Java Virtual Machines).

"The values returned by this method become meaningful only when the difference between two such values, obtained within the same instance of a Java virtual machine, is computed." - [20].

Tydligen fungerar det inte även om man försöker synkronisera de båda datorernas tider. Enligt slutsatsen i [21] bör man använda System.nanoTime() när man vill räkna ut tidsskillnader och system.currentTimeMillis om man vill ha den absoluta tiden men eftersom System.nanoTime inte funkar att synkronisera måste man försöka mäta tiden från samma källa. Istället för att försöka synkronisera tiden kan man mäta tiden mellan varje tidsstämpel i de mottagna paketen för UDP. För TCP uppkom istället ett annat problem. Det är svårt att dela in TCP:s paket på applikationsnivå, detta eftersom TCP själv delar in paketen i paket och skickar dem när det passar. Med andra ord har man som programmerare inte mycket kontroll över vad som händer när man väl skickat ut data till bufferten. Problemet är att det då blir svårt att hämta ut tiden ur paketen eftersom hela paketet som skickas inte kommer fram samtidigt. Mäta tiden kan man istället göra hos sändaren när all data har skickats på ett väldigt enkelt sätt. Genom att skicka ett litet ack-paket från mottagaren när all data har kommit fram och hos sändaren mäta tiden från att man börjar skicka tills att det ack-paketet mottagits. Fördelen med att göra på detta vis är dels att ack-paketet kan vara så litet som möjligt och att länken mellan de båda datorerna är väldigt kort. Med andra ord kan man försumma tiden det tar att skicka ett sådant litet paket. Problemen ansågs nu vara lösta och det var dags att göra mätningen för de båda protokollen på datorerna.

När mätningen sedan gjordes på detta sett visade det sig bli en stor skillnad mellan tiderna på de båda protokollen, mätningen för TCP blir dessutom inte så noggrann som den blir för UDP, men tidsskillnaden blev stor och paketförlusten blev väldigt stor för UDP.

Erbjuden trafik	Mottagen data UDP	Mottagen data TCP	Sänd-tid TCP (nanosekunder)	Sänd-tid UDP (nanosekunder)
10 000	10 000	10 000	130 000	36 904
100 000	100 000	100 000	33 518	13 500
1 000 000	540 000	1 000 000	19 056	-

Figur 25. En tabell som visar mätningen där paketstorleken är 1400 bytes. Mängden mäts i bytes.

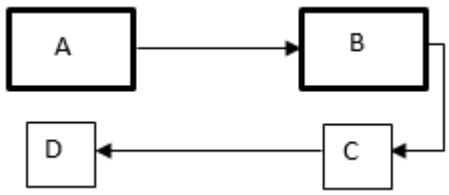
Paketförlusten för UDP beror med största sannolikhet på att buffertstorleken är för liten hos mottagaren och att för mycket logik ligger hos mottagartråden. Att sänd-tiden för varje paket blir lägre när en större mängd data skickas var oväntat. Det är mer logiskt att ett större antal paket som skickas på en gång får en högre fördröjning eftersom de hamnar i en längre kö i mottagarbufferten. Mätningen blir samtidigt inte särskilt rättvis då UDP mäter tidsskillnaden mellan varje paket och TCP mäter tidsskillnaden genom att ta RTT/2. Det bästa hade varit att kunna mäta tiden för de båda protokollen på samma sätt, så att man vet att skillnaderna mellan de båda protokollen inte har något att göra med hur tiden mäts.

Försök 6

Den höga paketförlusten beror med största sannolikhet på att buffertstorleken är för liten. I försök 2 visade det sig att det inte gick att höja buffertstorleken genom att ändra i filen sysctl.conf. Det finns även kommandon för att göra detta direkt i terminalen men det ska inte göra någon skillnad mer än att ändringen blir permanent ifall man skriver direkt i sysctl.conf. Efter ett flertal försök att ändra buffertstorleken direkt i terminalen visade det sig att följande kommando fungerar: sysctl -w net.core.rmem_max=1000000000. Detta ändrar buffertstorleken temporärt och måste göras varje gång datorn startats och som root-användare. Det gick heller inte att sätta buffertstorleken högre än 599000000000 byte (599 Gigabyte).

Det är dock inte rekommenderat att sätta en så stor buffertstorlek eftersom det kan orsaka alla möjliga sorters systemfel. Det går även att ändra sändarens buffertstorlek genom att skriva: sysctl –w net.core.wmem_max=1000000000. Detta räcker dock inte utan man måste även ändra buffertstorleken i Java när man skapar sina sockets genom att anropa socket.setReceiveBufferSize() eller socket.setSendBufferSize().

För att lösa tidmätningen för de båda protokollen bör det vara lämpligast att mäta tiden hos de båda protokollen på ett liknande sätt. För att mottagartråden ska bli så effektiv som möjligt bör enbart tråden ta emot paket och inget annat. Det man kan göra är att låta en annan tråd behandla paketen, d.v.s. hämta ut tiden ur paketen och beräkna tidsskillnaden. Men att mäta tidsskillnaden mellan varje mottaget paket går enbart om alla paket mottas i rätt ordning. Lösningen kan man illustrera enligt figur 26.



Figur 26. A skickar ett paket till B. B lämnar över paketet till C där det delas ner till ett mindre paket, sedan skickas det till D. D tar tiden när paketet mottogs subtraherat med tiden som finns i paketet, d.v.s tiden då det skickades från A.

Men hur ska paketen skickas mellan B och C? C behöver någon slags kö där den kan lagra paketen som den ska dela ner och skicka. För att göra detta kan man använda en ArrayBlockingQueue(). Detta är en FIFO (First In First Out) – kö som är generisk, vilket innebär att den kan lagra objekt av olika typer. Med hjälp av metoden take() i ArrayBlockingQueue(), kan man ta ut det första elementet i kön och behandla det. Är kön tom blockerar metoden tills ett element finns att hämta. En FIFO-kö är en slags kö där element lämnar kön i samma ordning som de läggs in i kön. Se [22] för mer detaljer.

Problemet var nu löst för UDP men ett problem kvarstod för TCP. TCP är ett strömbaserat protokoll till skillnad från UDP som är paketbaserat. Detta innebär att TCP själv delar in data som skickas i lämpliga paketstorlekar. Hos mottagaren kan man istället för en vanlig outputstream() använda sig av en dataOutputStream(). Anledningen till att man vill göra detta är för att kunna använda en metod som heter readFully(byte[] b). Denna metod blockerar tråden tills mängden data som motsvarar bytevektorn ”b” har lästs. Se [23] för mer information. På detta sätt kan man försäkra sig om att man alltid läser exakt ett pakets datamängd åt gången.

Eftersom buffertstorleken visade sig gå att ändra var det i sin ordning att bedöma hur man kan sätta buffertstorlekarna lika för de båda protokollen och samtidigt få bäst prestanda.

6. Resultat

Resultatet fokuserar på att visa mätresultatet av den sista och avgörande mätningen som görs på två datorer. Utifrån mätresultatet går det att ta fram en slutsats och därmed en slutsats för examensarbetet.

6.1. Mätresultat av implementation

Mätningen görs ett flertal gånger med tre olika paketstorlekar och med en noggrannhet på 50 körningar. Vid tidmätning tas ett genomsnitt som är likartat fördröjningen för varje skickat paket. Tanken med diagrammen är att se till vilken gräns man kan belasta TCP och UDP samt för att se vad som händer med de båda när de blir överbelastade. Mätningen har gjorts på två datorer med operativsystemet Linux Mint 17 där de båda datorerna har samma typ av nätverkskort med en kapacitet på 1Gbit/s och som båda är anslutna till varandra med en switch på kapaciteten 1Gibt/s. Protokollens kapacitet begränsas därför till en bandbredd på 1Gbit/s. Samma typ av mätning har även gjorts på HPC-klustret, men resultatet av denna mätning är inte tillgänglig för allmänheten. Resultatet av denna mätning går för Saabs del att läsa i bilagan: Teknisk rapport.

Först gjordes en mätning för att undersöka buffertstorlekarna.

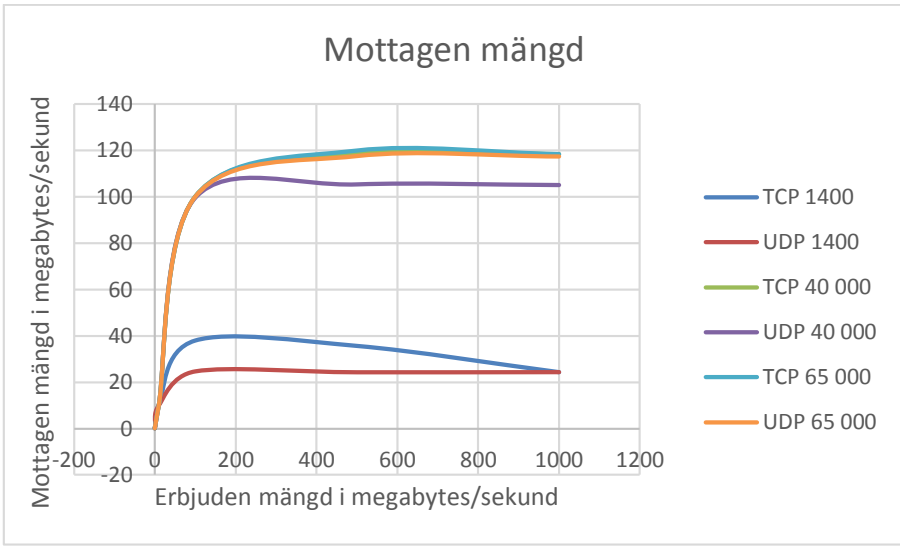
Sändbuffert	Mottagarbuffert	Mottagen data (TCP)	Mottagen data (UDP)	Sändningstid (TCP) ms	Sändningstid (UDP) ms
10 000 000	10 000 000	10 000 000	10 000 000	8.19	31.84
2240	10 000 000	10 000 000	10 000 000	0.25	0.27
10 000 000	2240	5 441 800	9 202 276	2.55	0.58
2240	2240	9 909 172	9 811 768	0.32	0.16

Figur 27. I denna tabell visas minsta och största buffertstorlekarnas påverkan på protokollen. Paketstorleken är 1400 bytes och alla datamängder i tabellen visas i bytes.

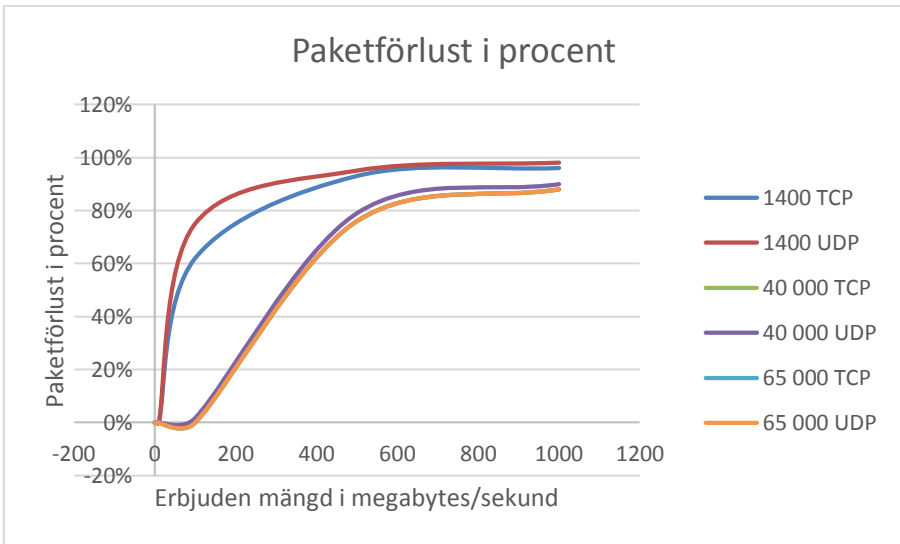
Sändbuffert	Mottagarbuffert	Mottagen data (TCP)	Mottagen data (UDP)	Sändningstid (TCP) ms	Sändningstid (UDP) ms
10 000 000	10 000 000	10 000 000	4 919 200	16.59	11.06
65 000	10 000 000	10 000 000	10 000 000	0.89	0.91
10 000 000	65 000	10 000 000	9 754 100	16.60	11.22
65 000	65 000	10 000 000	9 831 000	0.50	0.88

Figur 28. I denna tabell visas minsta och största buffertstorlekarnas påverkan på protokollen. Paketstorleken är 65 000 bytes och alla datamängder i tabellen visas i bytes.

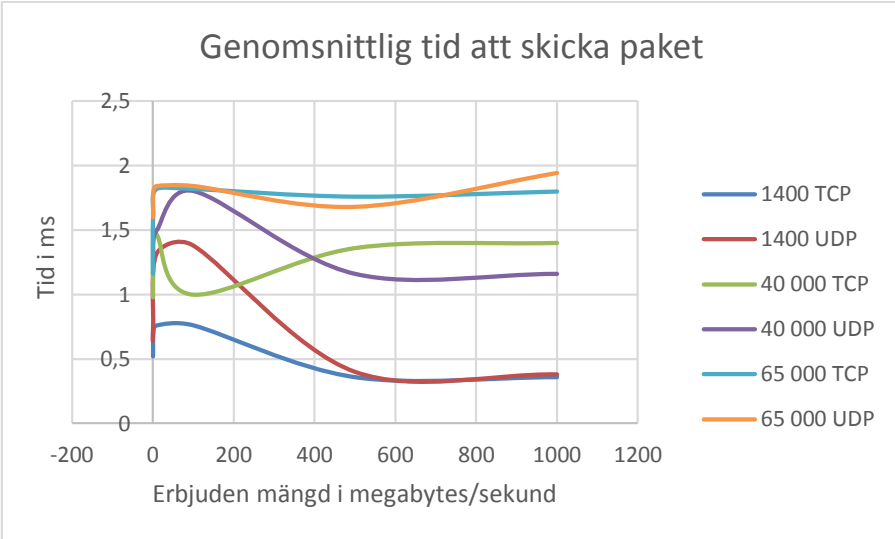
Som man kan se i figur 27 och 28 är den bästa gemensamma inställningen att ha en liten sändarbuffert och en stor mottagarbuffert för båda protokollen. Anledningen till att två mätningar gjorts är för att se om det går att hitta en gemensam buffertstorlek som passar båda protokollen och stora eller små paket. När buffertstorleken överstiger datorns tillgängliga ram-minne kommer en del av paketen att läggas direkt på hårddisken, vilket kan påverka resultatet men det är svårt att ta reda på hur stor påverkan det faktiskt gör. En liten sändarbuffert gör att paket inte behöver ligga i kö innan de skickas, vilket med största sannolikhet är anledningen till att sändningstiden för varje paket blev längre när sändarbufferten var stor, enligt figur 27 och 28. Som en naturlig följd hinner inte mottagaren ta emot all data som skickas på en sekund. I dessa mätningar avses paketförlusten vara den datamängd som inte hunnits skickas på en sekund.



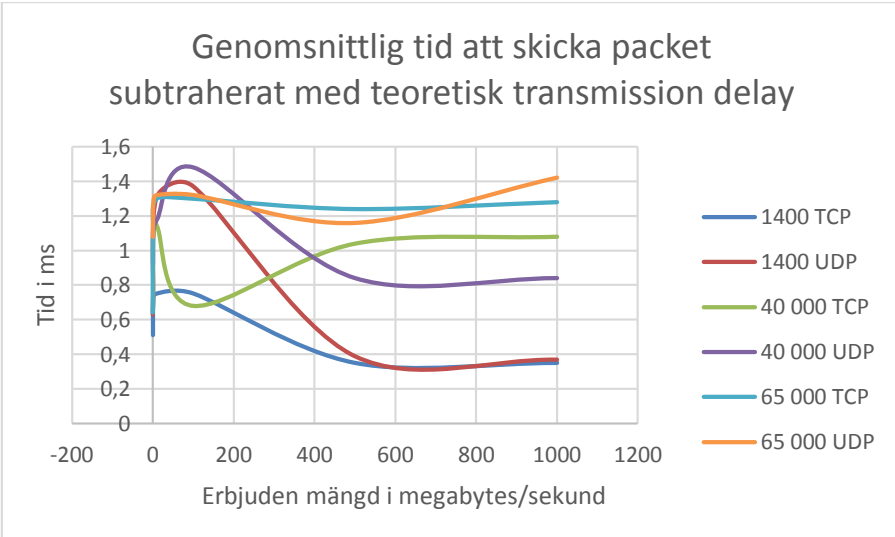
Figur 29. Graf som visar mottagen mängd i förhållande till erbjuden mängd.



Figur 30. Graf som visar paketförlusten i förhållande till erbjuden mängd.



Figur 31. Graf som visar den genomsnittliga tiden det tar att skicka ett paket i förhållande till erbjuden mängd data.



Figur 32. Transmission delay för paketen räknas ut genom att ta paketstorleken/länkkapaciteten. Figur 31 och 32 skiljer sig mer desto större paketen är. Med transmission delay avses den tid det tar för ett paket att överföras från bufferten till den fysiska länken.

Diskussion och kommentarer

De första två graferna i figur 29 och 30 ser ut så som man kan förvänta sig att de ska se ut. Den mottagna mängden ökar i takt med att den erbjudna mängden ökar tills att en viss gräns nås och på samma sätt ökar paketförlusten när denna gräns nås. Den högsta gränsen som nås är runt 120 mb/s vilket i bitar per sekund blir 960 Mbit/s. I början av detta avsnitt klargjordes det att båda datorernas nätverkskort samt switchen har en kapacitet på 1Gbit/s, vilket är länkens kapacitet. Denna bandbredd nås enbart när stora paket används eftersom bandbredden för de små paketen blir betydligt lägre för de båda protokollen. Detta är ett bra tecken, för det innebär att man kan nå upp till en bandbredd som i princip motsvarar länkens kapacitet.

Angående graferna i figur 31 och 32 finns det ett fenomen som man direkt lägger märke till. Den röda kurvan för UDP med paketstorlek 1400 bytes visar att det blir en fördröjning på ca 0.7 ms per paket när den erbjudna mängden är 100 megabytes men när den erbjudna mängden ökar, minskar fördröjningen och landar tillslut på ca 0.2 ms per paket. Samma sak händer även med TCP när paketstorleken är 1400 bytes men kurvan blir inte lika drastisk. Om man subtraherar den teoretiska transmission-delay (se figur 32) ser man att kurvan för UDP, med paketstorlek 1400 bytes, inte påverkas särskilt mycket. Detta kan bero på flera olika saker. Att kartlägga och ta reda på vad detta fenomen beror på hade varit intressant men det är inte inom ramarna av detta examensarbete. Några möjliga orsaker till att detta fenomen uppstår kan vara följande:

- Det kan ha att göra med hur operativsystemet schemalägger aktiviteter. Till exempel kanske mottagarbufferten töms med ett högre intervall när operativsystemet märker att intensiteten av paket ökar i mottagarbufferten.

- Det kan ha att göra med i vilken ordning paket tas ut ur mottagarbufferten. Om ett paket tas ut ur mottagarbufferten i en annan ordning än det ankom i, kan det orsaka en fördröjning för vissa paket. Om detta alltid händer för några paket vid varje sändningsintervall kan det förklara varför fördröjningen blir hög när ett litet antal paket skickas i varje intervall.
- Det är operativsystemet som bestämmer när paket skickas för både UDP och TCP samt när data plockas ner från mottagarbufferten, hur detta görs kan möjligtvis påverka tiden det tar att skicka paket.

Mätningen i sin helhet går att studera i de tabeller som finns i Appendix B.

7. Slutsats

Detta avsnitt handlar om vilka slutsatser man har kunnat ta baserat på resultaten i kapitel 6, författarens egna kommentarer om arbetet och vilka framtida utvecklingsmöjligheter som finns.

7.1. Kommentarer

Resultatet på den slutgiltiga mätningen var delvis vad man förväntade sig och delvis inte. Att den högsta bandbredden som uppnåddes var närliggande länkens kapacitet var ett positivt resultat, för det innebär att det inte finns något som begränsar prestandan. Det som dock var oväntat var att TCP faktiskt presterade lite bättre än UDP. Vad man egentligen förväntar sig är ju att UDP ska vara det snabbaste av de två protokollen. Med dessa resultat hade det varit intressant att se hur det blir när man skickar data från en sändare till flera mottagare. Man hade då kunnat använda multicast-protokollen som tillämpas under UDP och se hur det skiljer sig åt jämfört med att använda flera förbindelser med TCP.

7.2. Svar på frågeställningar

I detta avsnitt besvaras de frågeställningar som gjorde i början av arbetet. De flesta svaren baseras på det resultat som tagits fram i avsnitt 6. Med paketförlust avses här den mängd data som inte hunnit skickas på en sekund. Följande frågeställningar utformades i början av examensarbetet:

1. Vilket/vilka nätverksprotokoll är lämpligast för att transportera data i HPC-klustret?
2. Hur många processorkort behövs?
3. Hur mycket overhead kan man som mest tolerera?
4. Vilken metod är lämpligast att använda för att dela upp data mellan de olika processorkorten?

5. Hur mycket användbar data får man plats med på en länk i förhållande till länkens kapacitet?

6. Vilka andra tekniska problem finns med implementationen?

Följande frågeställningar utformades under arbetets gång:

7. Är det lämpligt att använda stora eller små paketstorlekar för att uppnå hög prestanda?
8. Hur påverkas tiden det tar att skicka ett paket?

Frågeställning 7 handlar om paketstorleken. Det visade sig under arbetets gång att paketstorleken var en viktig faktor som framförallt påverkar bandbredden. Detta går framförallt att se i figur 29 och 30. Frågeställning 8 utformades i samband med hur mätningarna skulle utföras. För att kunna se vad som händer med protokollen när de når den högsta bandbredden är det intressant att kolla på vad som händer med varje paket.

Följande svar har kunnat ges i slutet av arbetet och under arbetets gång:

1. Både TCP och UDP visar sig vara ungefär lika bra med bara några saker som skiljer dem åt. Paketförlusten är ungefär samma för de båda protokollen och beror på länkens kapacitet. Den genomsnittliga tiden för varje paket är i de flesta fall lägre för TCP än vad den är för UDP, vilket är oväntat eftersom man hade förväntat sig att UDP skulle vara snabbare. Detta innebär att TCP är det protokoll, utifrån resultatet, som vid en enskild förbindelse är bäst.
2. Har inte gått att besvara eftersom examensarbetet har fokuserat på att mäta prestandan mellan två noder och inte mellan en till flera noder. Se Avgränsningar i 1.4.
3. Denna frågeställning går inte att svara på helt och hållet eftersom overhead även till en viss del syftar på uppdelningsproblemet i avsnitt 2.5. Se Avgränsningar i 1.4.

4. Denna frågeställning går heller inte att svara på helt och hållet eftersom uppdelningsmetoderna även bör testas och utvärderas. Se Avgränsningar i 1.4.
5. Enligt figur 29 får man som mest plats med ungefär 96 % data i förhållande till länkens kapacitet på 1 Gbit/s.
6. Det finns väldigt många tekniska problem. De flesta tekniska problem som har uppstått beskrivs i avsnitt 5 och några i avsnitt 6. Generellt har de två största tekniska problemen varit hur man ska mäta tiden och hur man ska minska paketförlusten på applikationsnivå.
7. Enligt figur 29 och 30 är det mest lämpligt att använda så stora paket som möjligt. Stora paket ger en större bandbredd och mer data kan skickas per sekund.
8. I figur 31 ser man att den genomsnittliga tiden det tar att skicka ett paket planar ut i takt med att intensiteten av megabytes per sekund ökar. Det man också kan se är att tiden är större när intensiteten av megabytes per sekund är lägre. I figur 27 och 28 kan man se att tiden även påverkas av den buffertstorlek som sätts både hos sändaren och mottagaren.

7.3. Framtida utvecklingsmöjligheter

- Tillämpa multicast-protokollen och se hur se hur resultatet skiljer sig åt jämfört med att använda flera förbindelser med TCP
- Använda SCTP istället för TCP och se om prestandan blir bättre
- Ta reda på hur många processorkort i HPC-klustret som är mest effektivt att använda för en viss given mängd indata

Terminologi

UDP – User Datagram Protocol

TCP – Transport Control Protocol

IGMP – Internet Group Management Protocol

PIM – protocol Independant Multicast

RTP – Real-Time Transport Protocol

FTP – File Transfer Protocol

NTP – Network Time Protocol

SCTP – Stream Controlled Transport Protocol

Ack – acknowledgement, ett paket som skickas tillbaks till den som skickat ett paket för så att den som skickar vet att paketet kommit fram till mottagaren rätt.

RTT – Round trip time; Tiden det tar för att paket att skickas från en startpunkt till en slutpunkt och sedan tillbaks till startpunkten igen.

MSS – Most significant size; Den största storleken som går att ha på ett paket eller segment på ett visst nätverks-lager.

Referenser

[1]. James F Kurose; Keith W Ross; 2009-06-08, Computer Networking, Pearson Education, ISBN: 9780131365483.

[2]. Mathis M; Semke J; Mahdavi J; Ott T;, The macroscopic behaviour of the TCP congestion avoidance algorithm, Computer Communication Review; 07-1997, ISSN: 0146-4833.

[3]. Randall Stewart; Chris Metz; SCTP New Transport Protocol for TCP/IP, On the wire; 11-12 2001.

[4]. Daniel P. Meyer; Herbert A. Mayer; Radar Target Detection – handbook of theory and practice; 1973 Academic Press INC, ISBN: 0-12-492850-1

[5]. [https://technet.microsoft.com/en-us/library/bb742462.aspx] – *PIM SM Multicast Routing protocol*

[6]. RFC 3973

[7]. RFC 5015

[8]. RFC 3569

[9]. RFC 3286

[10]. RFC 5905

[11]. Javadoc – java.net

[12]. Javadoc – java.io

[13]. Official NTP Documentation; http://www.ntp.org/documentation.html; Network Time Foundation; senast hämtat: 2015-04-28

[14]. http://www.hpccsystems.com – HPC Kluster; senast hämtat: 2015-04-28

[15]. Neetu Goel; R.B.Garg, Ph.D; International Journal of Computer Applications(0975-8887); August-2013; Simulation of an optimum multilevel dynamic round robin scheduling algorithm

[16]. http://stackoverflow.com/questions/3755208/measuring-time difference-between-networked-devices/; Hämtad 2015-04-28

[17]. http://linux.die.net/man/1/nc – netcat; Hämtad 2015-05-15

[18]. RFC 896

[19]. http://en.wikipedia.org/wiki/Protocol_Independent_Multicast; Hämtad: 2015-05-15

[20]. https://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime() – System.nanoTime; Hämtad: 2015-05-21
[21]. https://blogs.oracle.com/dholmes/entry/inside_the_hotspot_vm_clocks/ - Blog på oracles hemsida som jämför system.nanoTime och system.currentTimeMillis; Hämtad 2015-05-27
[22]. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html/ - Javadoc för ArrayBlockingQueue; Hämtad 2015-05-28
[23]. https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html/ - Javadoc för DataInputStream; Hämtad 2015-05-28
[24]. Handledare på Saab
[25]. www.radartutorial.eu – grundläggande radarteori; Hämtad 2015-05-30
[26]. http://www.mellanox.com/ - Mellanox hemsida; Hämtad 2015-06-03
[27]. http://www-03.ibm.com/systems/platformcomputing/products/hpc/ - IBM:s hemsida med information om HPC-kluster; Hämtad 2015-06-03
[28]. http://skogberg.eu/ia/protstack.php/ - Källa till bild på TCP/IP stack; Hämtad 2015-06-03
[29]. http://www.bomara.com/ - Källa till bild på ett enkelt multicastnätverk; Hämtad 2015-06-03
[30]. http://www.ibm.com/developerworks/library/l-sctp/ - Artikel på IBM:s hemsida som handlar om SCTP; Hämtad 2015-06-03
[31]. RFC 2960 - SCTP
[32]. http://www.oracle.com/technetwork/articles/javase/index-139946.html - SCTP på Oracles hemsida; Hämtad 2015-06-03

Appendix A: Kod för Slutimplementation

TCP - Send.java

```
1.  import java.net.*;
2.  import java.nio.ByteBuffer;
3.  import java.util.Scanner;
4.  import java.util.Timer;
5.  import java.util.TimerTask;
6.  import java.io.*;
7.
8.  public class Send extends TimerTask{
9.      Socket sendSocket;
10.     BufferedOutputStream out;
11.     byte[] dataPacket, byteAddress;
12.     InetAddress adrs;
13.     static int dataSize, curDataSize, pktSize;
14.     long startTime, endTime, currentTime, packetCounter, tot
    alData;
15.     int portNbr, buffSize, packets, count;
16.     ByteBuffer bBuffer;
17.     static TimeCalculator tc;
18.
19.     public Send(int data, int packetSize, int port, int buff
    erSize){
20.         dataSize = curDataSize = data;
21.         pktSize = packetSize;
22.         portNbr = port;
23.         buffSize = bufferSize;
24.         packets = 0;
25.         bBuffer = ByteBuffer.allocate(8);
26.
27.         byteAddress = new byte[] {(byte)192, (byte)168, (byt
    e)0, (byte)2};
28.
29.         try {
30.             adrs = InetAddress.getByAddress(byteAddress);
31.             sendSocket = new Socket(adrs, portNbr);
32.             out = new BufferedOutputStream(sendSocket.getOut
    putStream());
33.             sendSocket.setSendBufferSize(buffSize);
34.             tc = new TimeCalculator();
35.         } catch (IOException e) {
```

```
36.             e.printStackTrace();
37.         }
38.     }
39.
40.     public byte[] createPacket(int pktSize){
41.         byte[] sendData = new byte[pktSize];
42.         ByteBuffer temp = ByteBuffer.allocate(8);
43.         currentTime = tc.getCurrentTime();
44.         temp.putLong(currentTime);
45.         temp.rewind();
46.         temp.get(sendData, 0, 8);
47.         curDataSize += pktSize;
48.         return sendData;
49.     }
50.
51.     public void closeConnections() {
52.         try {
53.             sendSocket.close();
54.             out.close();
55.         } catch (IOException e) {
56.             e.printStackTrace();
57.         }
58.     }
59.
60.     public long sentPackets() {
61.         return packetCounter;
62.     }
63.
64.     public long getTotalData(){
65.         return totalData;
66.     }
67.
68.     public void run(){
69.         try {
70.
71.             endTime = 0;
72.             if (!tc.isAlive()) tc.start();
73.
74.             while (curDataSize < dataSize) {
75.
76.                 startTime = System.nanoTime();
77.                 dataPacket = createPacket(pktSize);
78.                 out.write(dataPacket, 0, dataPacket.length);
```

```
79.                 out.flush();
80.                 endTime += System.nanoTime() - startTime;
81.                 packetCounter++;
82.                 totalData += dataPacket.length;
83.
84.                 if (endTime >= 1000000000) {
85.                     System.out.println("To much data to send
    for one second!");
86.                     System.out.println("total data sent!");

87.                     System.out.println("data sent: " + curDa
    taSize);
88.                     break;
89.                 }
90.             }
91.             System.out.println(count++);
92.             curDataSize = 0;
93.
94.         } catch (IOException e) {
95.             e.printStackTrace();
96.         }
97.     }
98.     public static void main(String[] args) {
99.         Scanner scan = new Scanner(System.in);
100.             int command, dataSize, portNumber, bufferSize,
    packetSize;
101.
102.             System.out.println("set data to send per secon
    d: ");
103.             dataSize = scan.nextInt();
104.             System.out.println("set portNumber: ");
105.             portNumber = scan.nextInt();
106.             System.out.println("set the size of the sendbu
    ffer: ");
107.             bufferSize = scan.nextInt();
108.             System.out.println("set the packetSize: ");
109.             packetSize = scan.nextInt();
110.
111.             Send client = new Send(dataSize, packetSize, p
    ortNumber, bufferSize);
112.             Timer timer = new Timer();
113.
114.             boolean run = true;
115.
```

```
116.             System.out.println("press 1 to start measure a
    nd 2 to quit...");
117.
118.             while (run) {
119.                 command = scan.nextInt();
120.                 switch(command) {
121.                 case 1:
122.                     timer.scheduleAtFixedRate(client, 0, 1
    000);
123.                     System.out.println("Sending...");
124.                     break;
125.                 case 2:
126.                     timer.cancel();
127.                     run = false;
128.                     System.out.println("Terminating...");

129.                     System.out.println("Total packets sent
    : " + client.sentPackets());
130.                     System.out.println("total sent data: "
    + client.getTotalData());
131.                     System.out.println("total packets rece
    ived: " + tc.getNbrOfPackets());
132.                     System.out.println("total average time
    : " + tc.getTotalAverageTime());
133.                     scan.close();
134.                     client.closeConnections();
135.                     break;
136.                 }
137.             }
138.         }
139.     }
```

TCP – Receive.java

```
1.  import java.net.*;
2.  import java.nio.ByteBuffer;
3.  import java.util.Scanner;
4.  import java.io.*;
5.
6.  public class Receive extends Thread {
7.      ServerSocket srvSocket;
8.      Socket socket;
9.      DatagramSocket dSocket;
10.     InetAddress adrs;
11.     DataInputStream bis;
12.     DatagramPacket packet, ackPacket;
13.     byte[] data;
14.     static int pktSize, dataTot;
15.     int port, buffer, nbrOfPackets;
16.     long totalData;
17.
18.     DataHandler dh;
19.
20.     /**
21.      *
22.      * @param totData
23.      * @param packetSize
24.      * @param bufferSize
25.      * @param portNumber
26.      */
27.     public Receive(int totData, int packetSize, int bufferSi
    ze, int portNumber) {
28.         pktSize = packetSize;
29.         dataTot = totData;
30.         buffer = bufferSize;
31.         port = portNumber;
32.         try {
33.
34.             dh = new DataHandler(pktSize);
35.
36.         } catch (SocketException | UnknownHostException e) {

37.             e.printStackTrace();
38.         }
39.     }
```

```
40.
41.     public void run() {
42.         try {
43.
44.             // Definera tcp-socket
45.             srvSocket = new ServerSocket(port);
46.             System.out.println("waiting for client to initia
    te measurement...");
47.             socket = srvSocket.accept();
48.             socket.setSoTimeout(10000);
49.             socket.setReceiveBufferSize(buffer);
50.             data = new byte[pktSize];
51.
52.             bis = new DataInputStream(new BufferedInputStrea
    m(socket.getInputStream()));
53.
54.             dh.start();
55.
56.             while (true) {
57.
58.                 bis.readFully(data);
59.                 totalData += data.length;
60.                 dh.addToQueue(data);
61.             }
62.
63.
64.         } catch (IOException | InterruptedException e) {
65.             e.printStackTrace();
66.         }
67.         System.out.println("totalData: " + totalData);
68.     }
69.
70.     public static void main(String[] args) {
71.         Scanner scan = new Scanner(System.in);
72.         int dataSize, portNumber, bufferSize, packetSize;
73.
74.         System.out.println("set data to receive per second:
    ");
75.         dataSize = scan.nextInt();
76.         System.out.println("set portNumber: ");
77.         portNumber = scan.nextInt();
78.         System.out.println("set the size of the receivebuffe
    r: ");
79.         bufferSize = scan.nextInt();
```

```
80.         System.out.println("set the packetSize: ");
81.         packetSize = scan.nextInt();
82.
83.         scan.close();
84.         Receive server = new Receive(dataSize, packetSize, b
    ufferSize,
85.                 portNumber);
86.         server.start();
87.     }
88. }
```

UDP – SendFile.java

```
1.  import java.net.*;
2.  import java.nio.ByteBuffer;
3.  import java.util.Scanner;
4.  import java.util.TimerTask;
5.  import java.util.Timer;
6.  import java.io.*;
7.
8.  public class SendFile extends TimerTask {
9.
10.     DatagramSocket socket;
11.     DatagramPacket packet;
12.     long dataLength;
13.     static long currentData, allData;
14.     long currentTime, startTime, sendTime;
15.     ByteBuffer bBuffer;
16.     byte[] sendByte;
17.     boolean running;
18.     InetAddress adrs;
19.     int sentData, portNbr, bufferSize, packetSize;
20.     int sequenceNumber, count;
21.     static TimeCalculator tc;
22.
23.     static Timer timer;
24.     static long totalData;
25.
26.     /**
27.      *
28.      * @param data - total data to send/second, unless time
    to send exceeds 1 second
29.      * @param port - receiver portnumber
```

```
30.      * @param packets - number of packets before ack has to
    be received
31.      * @param buffSize - size of the sendBuffer
32.      * @param packSize - the size of the packets to be sent
33.      */
34.     public SendFile(long data, int port, int buffSize, int p
    ackSize) {
35.         dataLength = data;
36.         portNbr = port;
37.         bufferSize = buffSize;
38.         totalData = 0;
39.         packetSize = packSize;
40.         sequenceNumber = 0;
41.
42.         try {
43.
44.             socket = new DatagramSocket();
45.             socket.setSendBufferSize(bufferSize);
46.             System.out.println("sendBuffert: " + socket.getS
    endBufferSize());
47.
48.             currentTime = startTime = 0;
49.
50.             //internetadress till servern (statisk)
51.             byte[] b = new byte[] {(byte)192, (byte)168, (by
    te)0, (byte)2};
52.             adrs = InetAddress.getByAddress(b);
53.
54.             // enbart fÃ¶r lokalt lÃ¤ge
55.             //adrs = InetAddress.getLocalHost();
56.
57.             tc = new TimeCalculator();
58.
59.         } catch (IOException e) {
60.             e.printStackTrace();
61.         }
62.     }
63.
64.     /**
65.      * Creates packets to send to receiver, depending on the
    remaining data
66.      * to send, the last packet might be smaller than the ot
    hers
67.      * @param pktSize - size of the packet
```

```
68.      * @param sequenceNbr - the current packetnumber in the
    sequence
69.      * @return packet to send as byteArray
70.      */
71.     public byte[] createPacket(int pktSize) {
72.
73.         byte[] packet = new byte[pktSize];
74.         bBuffer = ByteBuffer.allocate(8);
75.
76.         sendTime = tc.getCurrentTime();
77.
78.         bBuffer.clear();
79.         bBuffer.putLong(sendTime);
80.         bBuffer.rewind();
81.
82.         bBuffer.get(packet, 0, 8);
83.
84.         return packet;
85.     }
86.
87.     /**
88.      *
89.      * @return number of sent packets
90.      */
91.     public long getSequenceNbr() {
92.         return sequenceNumber;
93.     }
94.
95.     public void interruptSend(){
96.         running = false;
97.     }
98.
99.     /**
100.          * sends file to receiversocket, if elapsed time e
    xceed 1 second
101.          * loop will exit.
102.          */
103.         public void run() {
104.             try {
105.
106.                 running = true;
107.                 currentData = dataLength;
108.                 currentTime = 0;
109.                 totalData = 0;
```

```
110.                 if (!tc.isAlive()) {
111.                     tc.start();
112.                 }
113.
114.                 while (totalData < currentData && running)
    {
115.
116.                     startTime = System.nanoTime();
117.
118.                     sendByte = createPacket(packetSize);
119.                     packet = new DatagramPacket(sendByte,
    sendByte.length, adrs, portNbr);
120.                     socket.send(packet);
121.
122.                     totalData += packet.getLength();
123.                     allData += packet.getLength();
124.                     sequenceNumber++;
125.
126.                     currentTime += System.nanoTime() - sta
    rtTime;
127.
128.                     // limits the sender to send within 1
    second
129.                     if (currentTime >= 1000000000) {
130.                         System.out.println("to much data f
    or 1 second!");
131.                         System.out.println("skickad data:
    " + totalData);
132.                         running = false;
133.                         currentTime = 0;
134.                         break;
135.                     }
136.
137.                 }
138.                 totalData = 0;
139.                 System.out.println(count++);
140.                 //socket.close();
141.             } catch (IOException e) {
142.                 e.printStackTrace();
143.             }
144.         }
145.
146.         public static void main(String[] args) {
147.             Scanner scan = new Scanner(System.in);
```

```
148.             int command, dataSize, portNumber, bufferSize,
    packSize;
149.
150.             System.out.println("set data to send per secon
    d: ");
151.             dataSize = scan.nextInt();
152.             System.out.println("set portNumber: ");
153.             portNumber = scan.nextInt();
154.             System.out.println("set the size of the sendbu
    ffer: ");
155.             bufferSize = scan.nextInt();
156.             System.out.println("set the packet size: ");
157.             packSize = scan.nextInt();
158.
159.             boolean running = true;
160.             SendFile sf = new SendFile(dataSize, portNumbe
    r, bufferSize, packSize);
161.             timer = new Timer();
162.
163.             System.out.println("press 1 to start sending a
    nd 2 to stop the program...");
164.
165.             while (running) {
166.                 command = scan.nextInt();
167.                 switch (command) {
168.                 case 1:
169.                     timer.scheduleAtFixedRate(sf, 0, 1000)
    ;
170.                     System.out.println("sending..");
171.                     break;
172.                 case 2:
173.                     sf.interruptSend();
174.                     System.out.println("all data sent this
    session: " + allData);
175.                     System.out.println("skickade paket enl
    igt sendFile: " + sf.getSequenceNbr());
176.                     System.out.println("Antal paket enligt
    TimeCalculator: " + tc.getNbrOfPackets());
177.                     System.out.println("Genomsnittlig tid
    enligt TimeCalculator: " + tc.getTotalAverageTime());
178.                     timer.cancel();
179.                     tc.interrupt();
180.                     running = false;
181.                     scan.close();
```

```
182.                     break;
183.                 }
184.             }
185.         }
186.     }
```

UDP – ReceiveFile.java

```
1.  import java.net.*;
2.  import java.nio.ByteBuffer;
3.  import java.util.Scanner;
4.  import java.io.*;
5.
6.  public class receiveFile extends Thread{
7.
8.      DatagramSocket socket;
9.      DatagramPacket packet;
10.     byte[] buffer;
11.     ByteBuffer bBuffer;
12.     long totalData, currentTime;
13.     int portNbr, bufferSize, packetSize;
14.     static DataHandler dh;
15.
16.     /**
17.      *
18.      * @param port - port on which socket will be open
19.      * @param packets - number of packets to be received unt
    il ack-response
20.      * @param reciveBuffer - the size of the queue that stor
    es "un-received" packets
21.      * @param packSize - the size of the packets to send
22.      */
23.     public receiveFile(int port, int buffSize, int packSize)
    {
24.         portNbr = port;
25.         bufferSize = buffSize;
26.         packetSize = packSize;
27.     }
28.
29.     /**
30.      * creates a packet with the current JVM time in nanosec
    onds
31.      * @param - timestamp to put in packet
32.      * @return - packet to send as byteArray
33.      */
34.     public byte[] createTimePacket(long time) {
35.         currentTime = time;
36.         byte[] timePacket = new byte[8];
37.         ByteBuffer timeInBytes = ByteBuffer.allocate(8);
```

```
38.         timeInBytes.putLong(currentTime);
39.         timeInBytes.rewind();
40.         timeInBytes.get(timePacket, 0, 8);
41.         return timePacket;
42.     }
43.
44.     public void run() {
45.         try {
46.             socket = new DatagramSocket(portNbr);
47.             socket.setReceiveBufferSize(bufferSize);
48.             socket.setSoTimeout(10000);
49.             System.out.println("BufferSize: " + socket.getRe
    ceiveBufferSize());
50.             buffer = new byte[packetSize];
51.
52.             dh = new DataHandler();
53.
54.             while (true) {
55.
56.                 //skapar ett nytt mottagarpaket
57.                 packet = new DatagramPacket(buffer, buffer.l
    ength);
58.                 // hÃ¤mtar paketet frÃ¥n kÃ¶n
59.                 socket.receive(packet);
60.
61.                 dh.addToQueue(buffer);
62.
63.                 totalData += packet.getLength();
64.
65.             }
66.         }catch (IOException | InterruptedException e) {
67.             e.printStackTrace();
68.         }
69.         System.out.println("total received data: " + totalDa
    ta);
70.     }
71.
72.     public static void main(String[] args) {
73.         Scanner scan = new Scanner(System.in);
74.         int portN, buffSize, packSize;
75.
76.         System.out.println("set receiver port number: ");
77.         portN = scan.nextInt();
```

```
78.         System.out.println("set the receivers buffer size: "
    );
79.         buffSize = scan.nextInt();
80.         System.out.println("set the packetSize: ");
81.         packSize = scan.nextInt();
82.
83.         receiveFile rf = new receiveFile(portN, buffSize, pa
    ckSize);
84.         rf.start();
85.         System.out.println("Receiveing...");
86.
87.         scan.close();
88.
89.     }
90. }
```

TimeCalculator.java

```
1.  import java.net.*;
2.  import java.nio.ByteBuffer;
3.  import java.util.Scanner;
4.  import java.util.Timer;
5.  import java.io.*;
6.
7.  public class TimeCalculator extends Thread {
8.      private long receiveTime, sendTime;
9.      private DatagramSocket socket;
10.     private DatagramPacket packet;
11.     private byte[] buffer;
12.     private ByteBuffer b;
13.     private static long totalTime;
14.     private int nbrOfPackets;
15.
16.     /**
17.      *
18.      * @throws SocketException
19.      */
20.     public TimeCalculator() throws SocketException {
21.         socket = new DatagramSocket(8888);
22.         socket.setReceiveBufferSize(1000000);
23.         buffer = new byte[8];
24.         b = ByteBuffer.allocate(8);
25.         totalTime = 0;
```

```
26.         nbrOfPackets = 0;
27.     }
28.
29.     /**
30.      *
31.      * @return long med den genomsnittliga "send-
    time" fÃ¶r alla skickade paket.
32.      * om nbrOfPackets > 0, annars returnerar metoden 0.
33.      */
34.     public long getTotalAverageTime() {
35.         if (nbrOfPackets > 0) {
36.             return totalTime/nbrOfPackets;
37.         }
38.         return 0;
39.     }
40.
41.     /**
42.      *
43.      * @return antalet paket som har mottagits av denna trÃ¥
    d.
44.      */
45.     public int getNbrOfPackets() {
46.         return nbrOfPackets;
47.     }
48.
49.     /**
50.      * used to make sure that time is measured from the same
    source
51.      * @return current nanotime in nanoseconds
52.      */
53.     public long getCurrentTime() {
54.         return System.nanoTime();
55.     }
56.
57.     /**
58.      * Tar emot ett paket med tiden paketet skickades och rÃ
    ¤knar ut RTT/2
59.      * Tiden lagras i totalTime.
60.      */
61.     public void run() {
62.
63.         while (1>0) {
64.
65.             try {
```

```
66.
67.                 packet = new DatagramPacket(buffer, buffer.l
    ength);
68.                 socket.receive(packet);
69.                 receiveTime = System.nanoTime();
70.
71.                 nbrOfPackets++;
72.
73.                 b.clear();
74.                 b.put(buffer);
75.                 b.rewind();
76.                 sendTime = b.getLong();
77.
78.
79.                 totalTime += (receiveTime - sendTime);
80.
81.             } catch (IOException e) {
82.                 e.printStackTrace();
83.             }
84.         }
85.     }
86. }
```

DataHandler.java

```
1.  import java.io.IOException;
2.  import java.net.*;
3.  import java.nio.*;
4.  import java.util.concurrent.ArrayBlockingQueue;
5.  import java.util.concurrent.BlockingQueue;
6.
7.  public class DataHandler extends Thread {
8.      private BlockingQueue<byte[]> queue;
9.      private byte[] data, packetData;
10.     private ByteBuffer b;
11.     private final DatagramSocket socket;
12.     private DatagramPacket packet;
13.     private InetAddress adrs;
14.
15.     /**
16.      * Konstruktor
17.      *
18.      * @throws SocketException
```

```
19.      * @throws UnknownHostException
20.      */
21.     DataHandler(int packetSize) throws SocketException, Unkn
    ownHostException {
22.         queue = new ArrayBlockingQueue<byte[]>(10000);
23.         b = ByteBuffer.allocate(8);
24.         data = new byte[8];
25.         packetData = new byte[packetSize];
26.         socket = new DatagramSocket();
27.         socket.setSendBufferSize(1000000);
28.         byte[] adress = new byte[] { (byte) 192, (byte) 168,
    (byte) 0, (byte) 3 };
29.         adrs = InetAddress.getByAddress(adress);
30.     }
31.
32.     /**
33.      * LÃ¤gger till en bytevektor i kÃ¶n, om trÃ¥den Ã¤r dÃ¶
    d startas den pÃ¥ nytt
34.      *
35.      * @param b
36.      *             bytevektor frÃ¥n paketet hos mottagaren
37.      * @throws InterruptedException
38.      */
39.     public void addToQueue(byte[] b) throws InterruptedExcep
    tion {
40.         queue.put(b);
41.     }
42.
43.     /**
44.      * HÃ¤mtar paket frÃ¥n kÃ¶n och skickar ivÃ¤g dem till d
    en andra trÃ¥den Om kÃ¶n
45.      * Ã¤r tom dÃ¶r trÃ¥den.
46.      */
47.     public void run() {
48.         try {
49.             while (1 > 0) {
50.                 packetData = queue.take();
51.                 b.clear();
52.                 // hÃ¤mtar tiden frÃ¥n vektorn i kÃ¶n
53.                 b.put(packetData, 0, 8);
54.                 // lÃ¤gger in tiden i bytevektorn som ska sk
    ickas
55.                 if (b.getLong(0) > 0) {
56.                     b.rewind();
```

```
57.                     b.get(data);
58.                     // skickar ivÃ¤g paketet med tiden till
    den andra trÃ¥den
59.                     packet = new DatagramPacket(data, data.l
    ength, adrs, 8888);
60.                     socket.send(packet);
61.                 }
62.             }
63.
64.         } catch (IOException | InterruptedException e) {
65.             e.printStackTrace();
66.         }
67.     }
68.
69. }
```

Appendix B: Mätvärden

TCP med 1400 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	0,52
1	1	0,7
10	10	0,76
100	37,92	0,76
500	35,63	0,36
1000	24,3	0,36

UDP med 1400 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	0,64
1	1	0,76
10	10	1,32
100	24,62	1,38
500	24,24	0,4
1000	24,29	0,38

TCP med 40 000 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	0,98
1	1	1,5
10	10	1,46
100	100	1
500	118,9	1,36
1000	118,3	1,4

UDP med 40 000 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	1,24
1	1	1,42
10	10	1,5
100	99,4	1,8
500	105,2	1,16
1000	104,9	1.16

TCP med 65 000 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	1,16
1	1	1,6
10	10	1,82
100	100	1,82
500	119,9	1,76
1000	118,4	1,8

UDP med 65 000 bytes paketstorlek

Erbjuden Trafik: MB/s	Mottagen Data: MB/s	Sänd-tid/paket: ms
0,1	0,1	1,6
1	1	1,8
10	10	1,84
100	100	1,84
500	117,6	1,68
1000	117,4	1,94