Institutionen för kommunikation och information
Examensarbete i datalogi 20p (kursnamn+poäng)
C-nivå
Vårterminen 2005

Feltolerant dynamisk schemaläggning av kanaler på switchat Ethernet

Björn Oscarsson

HS-IKI-EA-05-103

Feltolerant dynamisk schemaläggning av kanaler på switchat Ethernet

Examensrapport inlämnad av Björn Oscarsson till Högskolan i Skövde för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information.

2005-06-05

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: _____

Handledare för examensarbetet: Henrik Grimm

Feltolerant dynamisk schemaläggning av kanaler på switchat Ethernet

Björn Oscarsson

Sammanfattning

Ethernet är den standard som används nästan överallt för nätverk i hemmen, på kontor och i industrier. Varken Ethernet eller switchat Ethernet är gjorda för att hantera realtidskommunikation då tiden det tar att skicka ett meddelande inte alltid sker inom en förutsägbar tid.

I denna rapport presenteras ett protokoll som tillåter feltolerant dynamisk schemaläggning av kanaler över ett switchat Ethernet. Protokollet är en vidareutveckling av tidigare arbete (ThrottleNet) som gjorts inom området där realtidstrafik ska garanteras över ett switchat Ethernet. Till skillnad från tidigare arbete använder sig detta protokoll av kritikaliteter för att explicit visa skillnaden mellan de kanaler som får tas bort och de kanaler som inte får tas bort. Kanalerna schemaläggs vid kortvariga överbelastningar för att optimera användningen av tillgängliga resurser baserat på hur värdefulla och kritiska kanalerna är för systemet.

Nyckelord: ThrottleNet, Realtidskommunikation, Distribuerade realtidssystem, Ethernet, Resursallokering, Bandbreddsgarantier.

Tack

Till min handledare Henrik Grimm som givit mig ovärderlig feedback genom hela projektet. Utan hans hängivenhet och intresse skulle mitt projekt inte blivit så intressant att hålla på med som det blev.

Innehållsförteckning

1 Introduktion **1**

2 Bakgrund **3**

2.1 Realtidssystem . . . 3
2.2 Distribuerade system . . . 3
2.2.1 Feltolerans . . . 4
2.3 Distribuerade realtidssystem . . . 5
2.4 Realtidskommunikation . . . 5
2.4.1 PAR-protokollet . . . 5
2.5 DeeDS . . . 6
2.5.1 Konsistensklasser . . . 6
2.5.2 Segment . . . 7
2.6 Ethernet/IEEE 802.3 . . . 7
2.6.1 Switchat Ethernet . . . 8
2.7 Schemaläggning av kommunikation . . . 9
2.8 ThrottleNet . . . 10
2.8.1 Centraliserad och decentraliserad topologi . . . 11
2.9 Decentraliserad resursallokering . . . 11
2.10 Schemaläggning i överbelastade realtidsdatabassystem . . . 12
2.10.1 Överbelastningsmodellen . . . 14
2.10.2 Arkitekturen . . . 14
2.10.3 Hantering vid överbelastning . . . 14

3 Problembeskrivning **16**

3.1 Mål . . . 17
3.2 Antagande . . . 17

4 Metod **18**

5 Distributed Resource Allocation Protocol **19**

5.1 Inkommande och utgående kanaler . . . 19
5.2 Gränssnitt . . . 20
5.2.1 Externt gränssnitt . . . 20
5.2.2 Internt gränssnitt . . . 20
5.3 Allokeringslistor . . . 22
5.4 Allokeringsprocessen . . . 23
5.4.1 Fas 1 . . . 23
5.4.2 Fas 2 . . . 24
5.4.3 Fas 3 . . . 26

5.5 Avallokeringsprocessen . . . 28
5.6 Hantering av begränsnings- och avallokeringsmeddelande . . . 28
5.7 Feltolerans . . . 29
5.7.1 Avallokeringsprocessen . . . 29
5.7.2 Inkommande och utgående begränsning/avallokering . . . 29
5.7.3 Allokeringsprocessen . . . 30
5.8 Schemaläggning vid överbelastning . . . 31
5.8.1 Värdefunktion . . . 31
5.8.2 Beräkning av sparad kapacitet . . . 33
5.8.3 Beräkning av utility loss . . . 34
5.8.4 Beräkning av utility loss density . . . 35
5.9 Förutsägbara allokerings- och avallokeringstider . . . 35

6 Diskussion **36**
6.1 Relaterade arbete . . . 36
6.2 Framtida arbete . . . 38
6.2.1 Förutsägbara allokerings- och avallokeringstider . . . 38
6.2.2 Användning av outnyttjad kapacitet . . . 38
6.2.3 Begränsningsmöjligheter i externa gränssnittet . . . 38
6.2.4 Dubbelriktade kanaler . . . 39
6.2.5 Broadcast/Multicast . . . 39
6.2.6 Hantering vid användning av flera switchar . . . 39
6.2.7 Förändring av parametrar i det externa gränssnittet . . . 40
6.2.8 Global optimering . . . 40
6.2.9 Förbättrad feltolerans . . . 40
6.2.10 Multipla nätverk . . . 41

Referenser **42**

1 Introduktion

Vanliga Ethernet-nätverk är inte gjorda för att hantera realtidskommunikation eftersom tiden det tar att skicka ett meddelande inte alltid är deterministisk. Orsaken till det icke-deterministiska beteendet är protokollet CSMA/CD som används då kommunikationslänken är delad av flera noder, dvs. hubbar används för att koppla samman noderna i nätverket. När kollisioner på nätverket inträffar kommer en återsändning av de paket som gick förlorade under kollisionen att ske. Denna återsändning sker på sådant sätt att sändningen upphör och efter en slumpmässig tid kommer sändningen att göras igen. Om en kollision inträffar igen kommer längsta tiden innan återsändning att vara dubbelt så lång som längsta tiden vid föregående återsändning.

En Ethernet-switch kommunicerar med *full-duplex* vilket innebär att trafik kan flöda i båda riktningarna samtidigt. Skillnaden mellan en hubb och en switch är att hubben bara skickar vidare all inkommande trafik till alla noder medan switchen delar upp kommunikationslänken, med hjälp av dess interna buffert, och skickar endast vidare till den tänkta mottagaren. Switchens flödeskontroll gör att kommunikationen blir oförutsägbar men kommer inte att användas förrän switchens interna buffert är full.

DeeDS är en distribuerad realtidsdatabas som utvecklas av forskargruppen för distribuerade realtidssystem vid Högskolan i Skövde. I dagsläget är DeeDS sammankopplat med hjälp av ett switchat Ethernet-nätverk och uppdateringen som sker mellan de olika replikorna kan inte ge några garantier för att en uppdatering sker inom en bestämd tid. Detta kan inte garanteras på grund av vad som tidigare nämnts i texten angående ett Ethernet-nätverks beteende. Beroende på vilken typ av konsistensklass den data som ska replikeras tillhör i DeeDS finns det olika typer av tidskrav. En av dessa konsistensklasser kallas för *bounded time* och kräver att all replikering ska ske inom en förutsägbar tid.

ThrottleNet är en lösning som byggs ovanpå ett vanligt switchat Ethernet och garanterar realtidstrafik genom att begränsa den trafik som varje nod genererar till en annan nod. Idag finns det två varianter av ThrottleNet, centraliserad och decentraliserad. Den centraliserade tillåter dynamisk allokering av kanaler[1], som begränsar utgående trafiken, medan den distribuerade varianten bara kan ha statiskt uppsatta kanaler från start. Mathiason och Amirijoo (2004) beskriver en lösning på en distribuerad resursallokering som bygger på two-phase commit (2PC). Där är det möjligt att dynamiskt skapa kanaler som skiljer sig med avseende på hur viktiga de är för systemet. Problemet med att använda sig av en algoritm som 2PC är att inblandade noder blir låsta under överenskommelsen då en kanal ska allokeras. Detta medför att andra allokeringsförsök under denna tid inte kommer vara möjliga att göra. Ett annat problem med 2PC är om nätverket blir partitionerat kommer noderna att vänta en godtyckligt lång tid tills ett beslut kan göras.

Denna rapport fokuserar på att ta fram ett protokoll som tillåter feltolerant dynamisk schemaläggning av kanaler över ett switchat Ethernet. Protokollet schemalägger dynamiskt kanaler med olika kapacitet och kritikalitet. Schemaläggningen som används kan även hantera kortvariga överbelastningar då mer kapacitet begärs än vad som finns tillgängligt. Dessa typer av överbelastningar löses beroende på vad för kritikalitet och

[1] En kanal är en virtuell/logisk enkelriktad koppling mellan en sändande nod och en mottagande nod.

utility kanaler har, som är ett mått på hur kritisk en kanal är för systemet, och begränsa eller ta bort de minst kritiska.

2 Bakgrund

Nedan presenteras olika delar som anses nödvändiga för att kunna förstå det problem som finns kring tidigare arbete samt målet för detta arbete. Det handlar om distribuerade system, realtidssystem och kommunikationen mellan noder i ett nätverk, även tidigare arbete samt arbete som på något sätt påverkar detta arbetet presenteras också.

2.1 Realtidssystem

Enligt Burns och Wellings (2001) finns det många olika tolkningar på vad ett realtidssystem exakt är. Mellan dessa olika tolkningar är definitionen på responstid gemensamma, dvs. tiden det tar för systemet att reagera på en händelse (*Oxford Reference Online*, 2005).

Definitionen av ett realtidssystem som används i denna rapport är ett system där tiden det tar att producera utdata är signifikant. Ofta beror detta på att indata är kopplat till någon rörelse i den fysiska världen, och att utdata måste vara relaterad till samma rörelse. Fördröjningen från tiden då indata uppstod till utdata måste vara tillräckligt liten för acceptabel punktlighet (*Oxford Reference Online*, 2005).

Detta innebär att tidsskillnaden måste vara tillräckligt liten så att inte några deadlines bryts för det totala systemet. Exempel på sådana system kan vara ett robotsystem eller en automatiserad tillverkningslinje för bilar (*Oxford Reference Online*, 2005). Realtidssystem kan oftast delas upp i två olika typer, hårda och mjuka realtidssystem. Hårda realtidssystem (Burns och Wellings, 2001) är sådana system där det är nödvändigt att ett svar fås inom en specifik deadline. Mjuka realtidssystem är sådana system där responstiderna är viktiga, dock kommer systemet kommer att fungera korrekt även om dessa typer av deadlines bryts ibland.

2.2 Distribuerade system

Större delen av de datorer som används är ihopkopplade på något sätt via någon typ av nätverk. Ett exempel på sådant nätverk är Internet som binder samman flera olika nätverk. Definitionen som används för ett distribuerat system i denna rapport är hårdvaru- eller mjukvarukomponenter som är belägna på datorer sammankopplade via ett nätverk där koordinering endast sker med hjälp av att skicka meddelanden (Coulouris m.fl., 2001).

Den huvudsakliga orsaken till att skapa och använda sig av ett distribuerat system kommer från behovet av att dela resurser (Coulouris m.fl., 2001), exempelvis en skrivare, filer, hemsida eller annan typ av data. Andra motiveringar till att använda sig av ett distribuerat system är för att öka tillförlitligheten och tillgängligheten på systemet (Elmasri och Navathe, 2000). Detta ökas genom att exempelvis distribuera ut systemet över flera olika noder och skapa redundans. Tillförlitligheten definieras ofta som sannolikheten att systemet körs, medan tillgängligheten är sannolikheten att systemet är tillgängligt under en viss period.

Datorer som är kopplade till ett nätverk behöver inte vara inom samma rum eller samma byggnad utan avståndet mellan datorerna i nätverket kan vara vilket som helst. Definitionen för ett distribuerat system som nämndes tidigare medför några viktiga konsekvenser (Coulouris m.fl., 2001):

- **Parallellism**
  I ett nätverk som kopplar samman ett antal datorer kommer det normala fallet att vara att exekvering kommer att ske samtidigt på de olika datorerna. Detta innebär att en person kan sitta och skriva i ett dokument på sin dator under tiden en annan sitter och skriver i ett dokument på sin dator och de båda delar resurser som t.ex. filer. Det viktiga är då att kunna hantera den samtidighet som finns i systemet.

- **Ingen gemensam klocka**
  När program ska göra en koordinerad aktion görs detta endast via meddelande som skickas mellan de olika datorerna, som nämndes tidigare. Koordineringen kan bli problematisk eftersom de påverkade datorerna ej har någon gemensam klocka som kan säga den exakta ordningen när händelserna inträffade. Synkronisering av datorernas klockor möjlig men kan ibland inte vara tillräckligt exakt. Om klockorna inte är tillräckligt exakta kan det bli svårt att avgöra den verkliga ordningen mellan händelserna när två eller flera händelser sker väldigt nära varandra.

- **Oberoende fel**
  Alla datorsystem kan krascha helt oberoende av varandra. Det kan också vara kommunikationslänken mellan datorerna i nätverket som kan orsakar fel eller sluta fungera. Detta kan skapa många nya typer av fel jämfört med system som är rent centraliserade. Det är viktigt att ta hänsyn till detta vid design av systemet och hur olika scenario som kan uppstå ska hanteras.

2.2.1 Feltolerans

Ett distribuerat system kan partiellt sluta att fungera, dvs. olika komponenter kan sluta fungera medan andra fortsätter att fungera (Coulouris m.fl., 2001). I och med att systemet kan partiellt sluta att fungera medför detta även att hanteringen av fel blir svårare. Vissa fel kan direkt detekteras medan andra kan inte detekteras alls. Ett fel som finns i systemet som inte kan detekteras utan bara misstänkas är en utmaning att hantera. När ett fel upptäcks är det möjligt att dölja felet genom att exempelvis skicka om meddelandet, som inte kom fram till den tänkta mottagaren. Att dölja fel för användaren är inte alltid så praktiskt i stora nätverk som exempelvis Internet. Istället måste systemet tolerera fel men kräver att användaren gör detta också. När exempelvis en webbläsare kontaktar en webbserver som inte svarar kommer några försök att göras innan användaren får reda på problemet. När en komponent i ett distribuerat system slutar att fungera påverkas endast de som använder denna komponent. Om exempelvis en webbserver går ner kan denna tjänst tillhandahållas av en annan server och användarna kan använda den nya servern så länge. En klassificering av de olika fel som kan uppträda kan göras enligt följande punkter (Coulouris m.fl., 1994):

- **Försummelsefel (Omission failure)**
  När en server struntar i att svara vid begäran.

- **Svarsfel (Response failure)**
  När en server ger ett felaktigt svar vid begäran.

- **Tidsfel (Timing failure)**
  När en server inte svarar på begäran inom ett visst tidsintervall.

- **Kraschfel (Crash failure)**
  När en server inte svarar på flera begäran.

- **Slumpmässigt fel (Arbitrary failure)**
  Alla ovanstående fel.

2.3 Distribuerade realtidssystem

När det kommer till distribuerade system sker exekvering genom att varje process själv får exekvera lokal data och ta lokala beslut (*Oxford Reference Online*, 2005). Processerna byter ut information mellan varandra över ett nätverk för att bearbeta data eller för att läsa beslut som påverkar flera processer.

I ett distribuerat realtidssystem måste allt ske inom en förutsägbar tid. Varje nod för sig har lokala tidskrav på hur lång tid exekvering får ta samt att det är distribuerat över flera noder. Det totala systemet kan även ha globala tidskrav som ställer krav på att kommunikationen mellan noderna också sker inom en förutsägbar tid (Kopetz, 1997).

2.4 Realtidskommunikation

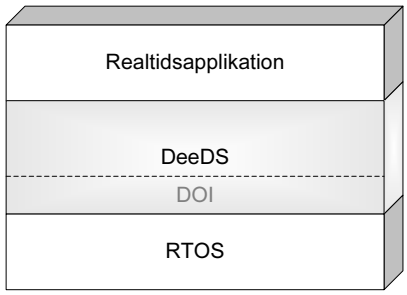
Realtidskommunikation är när ett meddelande överförs från en sändande nod till en mottagande nod inom ett förutsägbart tidsintervall med lite jitter och hög pålitlighet (Kopetz, 1997). Jitter är en en oönskad variation som förekommer då olika typer av fel finns i systemet vilket medför att den totala exekveringstiden förlängs. Kraven som finns på realtidskommunikation är att tiden det tar att skicka från en sändande nod till att den mottagande noden får meddelandet ska vara låg med minimalt jitter samt att feldetektionen bör vara snabb hos mottagaren.

2.4.1 PAR-protokollet

Positive-Acknowledgment-or-Retransmission (PAR) är en välkänd explicit flödeskontroll som är händelsestyrd (Kopetz, 1997). Händelsestyrd innebär att kommunikation eller någon typ av bearbetning initieras när en betydande förändring noterats. Explicit flödeskontroll är när mottagaren sänder ett explicit meddelande till sändaren för att informera sändaren om att meddelandet mottagits korrekt och att mottagaren är redo att acceptera nästa meddelande. PAR-protokollet innehåller en timeout och en försöksräknare som räknar antalet gjorda försök. När klienten till en sändare vill skicka ett meddelande återställer sändaren försöksräknaren till noll och startar dess lokala timeout. Meddelandet skickas iväg via kommunikationslänken till mottagaren. När sändaren mottagit bekräftelse från mottagaren inom den angivna timeouten informeras klienten att överföringen lyckades. Om sändaren inte får någon bekräftelse innan timeouten har gått ut kommer sändaren att kontrollera dess försöksräknare för att se om maximalt antal försök redan är gjorda eller ej. Om så inte är fallet kommer försöksräknaren att räknas upp med ett, timeouten startas om och ett nytt meddelande skickas till mottagaren. Skulle det vara så att maximala antalet försök redan är gjorda avbryts kommunikationen och klienten informeras om att överföringen misslyckades. Om mottagaren får ett meddelande som den redan mottagit kommer bara en ny bekräftelse att skickas tillbaka till sändaren.

2.5 DeeDS

Distributed Active Real-Time Database Systems (DeeDS) är en distribuerad realtidsdatabas som utvecklas av forskargruppen för distribuerade realtidssystem (DRTS) vid Högskolan i Skövde. Det tänkta användningsområdet för DeeDS är komplexa realtidssystem som kräver distribution och har en större mängd data som är gemensam med full replikering av databasen (Andler m.fl., 1996). Exempel på sådant system kan vara ett fordons kontrollsystem som drivs av autonoma noder, kontrollerar individuella subsystem samt hanterar stor mängd data under hårda tidskrav, ex. bränsleinsprutning, samtidigt som data kommer från andra mindre tidskritiska subsystem som växellåda eller olika typer av sensorer.



Figur 1: DeeDS arkitektur, omarbetning efter Andler m.fl. (1998).

En förenklad version av DeeDS arkitektur visas i figur 1 där DeeDS är byggt ovanpå ett realtidsoperativsystem (RTOS). Genom att använda sig av abstraktionslagret *DeeDS Operating System Interface* (DOI) skyddas alla funktionerna i DeeDS från operativsystemet (Andler m.fl., 1998). Detta gör det möjligt att på ett lättare sätt flytta DeeDS till andra operativsystem genom att endast göra förändringar i DOI. Kommunikationen mellan DeeDS noder är idag uppbyggd endast med en vanlig Ethernet switch där TCP/IP används som transportprotokoll och gör att uppdateringar inte kan ske inom en förutsägbar tid eftersom Ethernet inte är deterministiskt, se sektion 2.6.

2.5.1 Konsistensklasser

Applikationer har olika krav på den *ömsesidiga konsistensen*[2] mellan replikorna i systemet (Mathiason, 2002). Detta kallas för konsistensklasser och kan användas för att schemalägga replikering . Dessa konsistensklasser existerar för att stödja olika konsistenskrav i systemet.

Mathiason (2002) beskriver tre olika konsistensklasser med varierande realtidsegenskaper:

- **Immediate consistency**
  Denna klass innebär att replikorna som finns alltid är ömsesidigt konsistenta med varandra oberoende av tidpunkten. En replika kommer inte visa dess uppdateringar förrän alla andra replikor har gjort uppdateringen.
- **Eventual consistency**
  Denna klass innebär att replikorna utför förändringen lokalt och replikerar

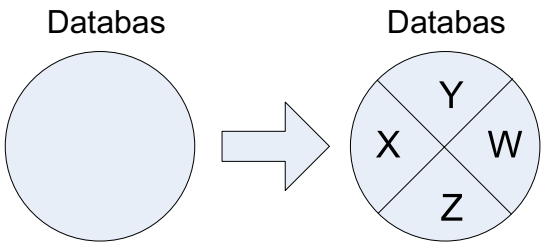
[2] Replikor som är konsistenta med varandra (Mathiason, 2002).

uppdateringen till replikorna. Så fort uppdateringen har gjorts är den synlig lokalt. Detta innebär också att replikeringen av uppdateringen kommer att ske efter att den lokala noden har hunnit att göra en commit på transaktionen. Eftersom parallell uppdatering av replikor tillåts är det också viktigt att kunna upptäcka och lösa konflikter som kan ske. Detta innebär också att applikationen måste tillåta att det temporärt kan uppstå inkonsistens i systemet.

- **ASAP**
  Replikering av uppdateringar sker inom en oförutsägbar tid men lokalt sker uppdateringen fortfarande förutsägbart.
- **Bounded time**
  Replikering av uppdateringar sker inom en förutsägbar tid och denna tid inkluderar konflikthantering.

2.5.2 Segment

Ett segment i en distribuerad realtidsdatabas är en grupp dataobjekt som delar samma egenskaper (Mathiason, 2002).



Figur 2: Segmentering av databasen, omarbetning efter Mathiason (2002).

Figur 2 visar en segmentering av en databas till fyra segment. Dessa fyra olika segment har olika typer av egenskaper och de dataobjekt som tillhör ett visst segment får också segmentets egenskaper. Ett segment behöver inte finnas på samtliga noder som existerar i systemet utan kan finnas på bara en delmängd av alla noder. Genom detta kan graden av replikering anges om exempelvis segmentet ska replikeras till alla noder eller bara några få. När varje segment bara replikeras till de noder som är i behov av segmentets data så är databasen *fullt virtuellt replikerad*. Det finns även andra typer av egenskaper ett segment kan ha, där en av de viktigaste är vilken typ av konsistensklass segmentet tillhör. Dynamisk allokering av segment är något som är viktigt att ha när tillgängligheten av data förändras med tiden (Mathiason, 2002).

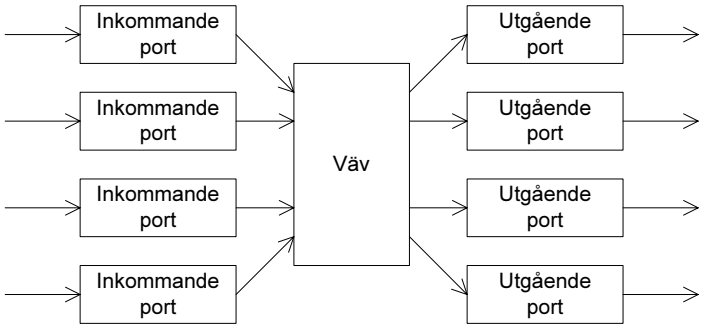
2.6 Ethernet/IEEE 802.3

Ethernet, IEEE 802.3, är den standard som används nästan överallt för nätverk i hemmen, på kontor och i industrier. Ethernet har genomgått många olika faser under dess utveckling sedan den först introducerades men bygger fortfarande på samma grundläggande idéer (Halsall, 2001). I äldre LAN som är ihopkopplade med hubbar och där CSMA/CD används sker kollisioner ofta och medför att den teoretiska maximala hastigheten ej kan nås. Genom att använda sig av switchar istället för hubbar används inte CSMA/CD. Då används switchens egna flödeskontroll som endast utnyttjas när dess interna buffert blir full, vilket ger ungefär lika dåliga

deterministiska egenskaper som CSMA/CD. När kollisioner upptäcks då CSMA/CD används kommer den sändande noden att skicka ut en blockeringssekvens och sedan vänta en slumpmässig tid innan ett nytt försök görs (Halsall, 2001). Om nästkommande försök också skulle misslyckas kommer tiden tills nästa försök att ökas exponentiellt av en schemaläggningsprocess som kallas för *truncated binary exponential backoff*. Denna nästkommande tid som schemaläggningsprocessen beräknar kallas för "backoff" (Halsall, 2001). En maximal gräns för hur många försök som får göras är satt och om denna gräns nås kommer den data som skickades inte att skickas igen. Detta beteendet som CSMA/CD har gör att kommunikationen mellan noderna inte blir deterministiskt.

2.6.1 Switchat Ethernet

I ett Ethernet-nätverk som inte är switchat är det omöjligt för två noder att samtidigt sända data som motsvarar dess maximala gräns eftersom kommunikationslänken är delad (Peterson och Davie, 2000).



Figur 3: 4 x 4 switch, omarbetning efter Peterson och Davie (2000).

Alla switchar, oavsett om det är ett switchat Ethernet eller ej, kan konceptuellt beskrivas som en hårdvara som har flera in- och utgångar (portar) som binds samman med ett slags väv (*fabric*) (Peterson och Davie, 2000), se figur 3. Väven i detta fall skapar kopplingarna från en eller flera ingångar till en eller flera utgångar. Switchens huvuduppgift är att ta emot inkommande paket och skicka iväg dessa till andra kommunikationslänkar som är kopplade till switchen. Denna funktion kallas för *switching* eller *forwarding*. I switchade Ethernet-nätverk använder switcharna sig av endast en stor minnesbuffert. Denna buffert delas upp mellan de in- och utgångar som finns och kan innehålla flera ramar[3] vilka antingen väntar på att bli bearbetade (inkommande) eller ivägskickade (utgående) och som lagras som en FIFO-kö (Halsall, 2001). Om hastigheten för inkommande ramar som ska till en utgående port överskrider hastigheten på den utgående porten kommer det uppstå något som kallas *contention* (Peterson och Davie, 2000), vilket innebär att det sker en tävlan om vem som ska få tillgång till resursen (*Oxford Reference Online*, 2005). Switchen kommer då att börja köa upp ramar som inte kan skickas direkt. När buffertutrymmet för utgående porten tar slut kommer det medföra att switchen måste börja kasta bort de ramar som inte får plats. När bortkastningen av ramar sker så ofta att nätverkets genomströmning försämras kallas detta *network congestion* (*Wikipedia*, 2005), då switchen blivit *congested*.

Om switchen har *n* ingångar där varje kommunikationslänk klarar av hastigheten s_n, så blir genomströmningen summan av alla s_n. Detta skulle vara den teoretiskt optimala

[3] En ram är ett block data (eng. frame) (*Oxford Reference Online*, 2005).

genomströmningen för switchen men kan inte garanteras i verkligheten eftersom det finns mer faktorer som påverkar (Peterson och Davie, 2000). Om det under en period då alla ingångar på switchen tar emot trafik med dess maximala bandbredd och all trafik ska till en och samma utgång kommer trafiken som inte kan skickas direkt att buffras eller kastas bort, beroende på om det finns något ledigt minne i bufferten. Detta sker eftersom summan av ingångarnas bandbreddsanvändning kommer att överstiga den maximala bandbredden som utgången kan skicka med. Under denna period är det inte möjligt att hålla en högre genomströmning än vad utgången klarar av. Det finns en möjlighet att hålla denna maximala hastighet om all inkommande data från ingångarna distribueras ut jämnt över alla utgångarna. Andra faktorer som är viktiga att räkna in är också switchens prestanda. Ethernet nätverk kan skicka data med varierande storlek eftersom varje ram inte har en fast ramstorlek, det kan även förekomma en viss grad av intern tävlan om resurser i switchen. Eftersom det inte finns någon fast ramstorlek kommer det ta olika lång tid att behandla inkommande data beroende på storlek (Peterson och Davie, 2000). För att minimera denna faktor måste antalet olika ramstorlekar minskas eller göras konstant.

2.7 Schemaläggning av kommunikation

När kommunikation sker mellan processer på olika noder kommer den data som ska skickas göra att det uppstår en slags tävlan om vilken lokal process som får i väg sin data först ut på nätverket (Burns och Wellings, 2001). För att kunna ha hårda realtidsprocesser som håller sina deadlines krävs det att accessen till kommunikationslänken schemaläggs i en ordning så att den överensstämmer med den schemaläggning av processer på varje processor. Standardprotokoll som är kopplade till Ethernet kan inte garantera hård realtidstrafik eftersom oförutsägbara *back-off* algoritmer används eller för att ordningen som används är FIFO.

Burns och Wellings (2001) skriver att en kommunikationslänk är som vilken annan resurs som helst. De nämner också tre punkter som skiljer problemen i schemaläggning av kommunikation från vanlig schemaläggning på en processor:

- **Flera åtkomstpunkter**
  Åtkomsten av en processor kan bara ske på ett sätt medan i ett nätverk där flera noder är i hopkopplade har en nod flera åtkomstpunkter. För att hantera detta krävs någon typ av distribuerat protokoll.

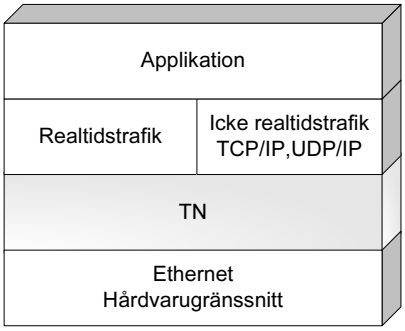
- **Preemption**
  Preemption fungerar bra vid schemaläggning där endast en processor finns. Om det skulle ske preemption på schemaläggningen av en kanal kommer det medföra att data som endast några fragment hunnit skickats av när preemption görs måste skickas om helt.

- **Deadlines påverkas inte bara av applikationen**
  De deadlines som måste hållas påverkas bara av själva applikationen när endast en processor används. När flera processorer blir inblandade kommer även deadlines påverkas av tillgängligheten på bufferten då data måste skickas iväg innan bufferten kan fyllas igen.

2.8 ThrottleNet

De första idéerna kring ThrottleNet presenterades i Martinsson (2002). ThrottleNet är en realtidsteknologi som är baserat på ett switchat Ethernet med *commercial off-the-shelf* (COTS) switchar, dvs. switchar som kan köpas i vilken datorbutik som helst (Blomdell m.fl., 2004). Med ThrottleNet kan en övre gräns för nätverkets fördröjning bestämmas genom att begränsa nätverkstrafiken, tekniken kallas för *throttling* eller *traffic smoothing*. Det fungerar så att varje nod begränsar sin utgående trafik för att förhindra att den totala trafiken på nätverket ej överskrider den maximala gräns som en switch klarar av på varje port innan det uppstår trafikstockning. Denna begränsning görs med riktade kanaler som måste skapas innan en nod kan börja skicka data till en annan nod. Det finns två typer av kanaler, realtidskanaler och icke realtidskanaler. Varje kanal är associerad med en period och en maximal ramstorlek. Perioden är det minimala intervall som får vara mellan två efterföljande sändningar på en kanal men ThrottleNet stöder även sporadisk kommunikation, inte bara periodisk. Med sporadisk kommunikation syftar det till kommunikationen sker inom ej bestämda intervall medan periodisk kommunikation sker inom bestämda intervall (*Oxford Reference Online*, 2005). Kanalens maximala ramstoleken är den samma som Ethernets maximala ramstorlek eftersom om kanalens maximala ramstorlek skulle vara större än Ethernets maximala ramstorlek skulle det innebära att kanalens meddelande var tvungen att fragmenteras varje gång data ska skickas.

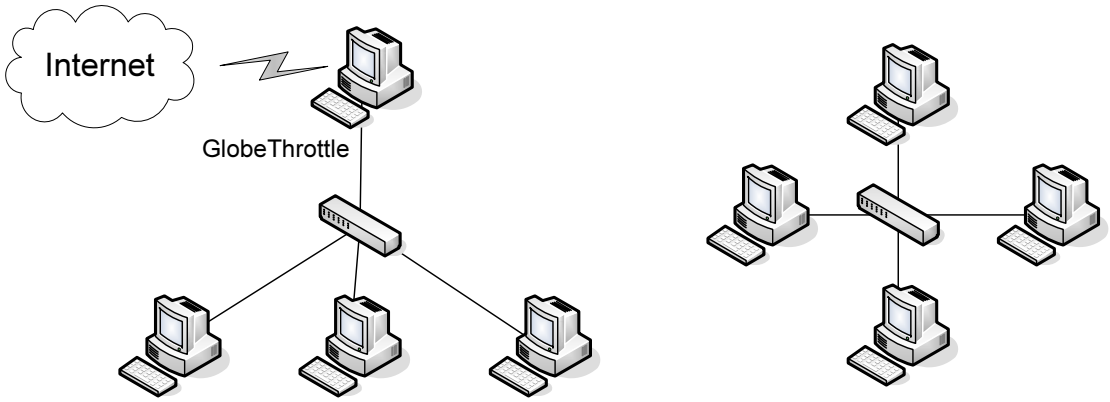
En icke realtidskanal är en kanal som hanterar nätverkstrafik som exempelvis UDP- eller TCP/IP-trafik. Dessa kanaler fungerar på samma sätt som realtidskanaler som beskrivs ovan men för denna typ av trafik är tiden det tar för att sända data från en sändarnod till en mottagarnod inte begränsad.



Figur 4: ThrottleNets arkitektur, omarbetning efter Blomdell m.fl. (2004)

ThrottleNet implementeras som ett lager direkt ovanför Ethernet som visas i figur 4. Detta lager kallar Blomdell m.fl. (2004) för *thin layer* (TN) och implementerar två huvudfunktioner: *trafikbegränsning* och *fragmentering/-defragmentering*. Fragmenteringsfunktionen används i huvudsak till icke realtidstrafik som exempelvis UDP datagram då ramens storlek kommer att bli större än Ethernets maximala ramstorlek. Detta kan ske eftersom extra information läggs till i TN-lagret om vilken typ av trafik det är.

2.8.1 Centraliserad och decentraliserad topologi



Figur 5: Vänstra figuren visar den centraliserade topologin, till höger visas den decentraliserade topologin, omarbetning efter Blomdell m.fl. (2004).

Blomdell m.fl. (2004) beskriver två metoder för hur ThrottleNet kan användas, centraliserat eller decentraliserat (se figur 5). Med det centraliserade topologin finns det en nod som kallas för GlobeThrottle som agerar som en nätverksvakt och implementerar schemaläggningsanalys. Skapande och borttagning av realtidskanaler kan ske dynamiskt. En sändarnod måste kommunicera med GlobeThrottle genom en icke realtidskanal när den vill upprätta en ny kanal och måste vänta på ett svar från GlobeThrottle om det är möjligt. Eftersom kommunikationen med GlobeThrottle sker via en icke realtidskanal kommer det inte finnas någon övre gräns för hur lång tid det kommer ta innan den nya kanalen är allokerad. Centraliserad topologi tillåter både realtidstrafik och icke realtidstrafik att köras på samma nätverk då GlobeThrottle är den enda noden som är kopplad till andra nätverk som exempelvis Internet som visas i figur 5. Med decentraliserad topologi tillåts inte någon icke realtidstrafik och nätverket är isolerat från andra nätverk som visas i figur 5. Dessutom är alla noder identiska och kanaler upprättas statiskt innan systemet startas, dvs. under körning får ej någon kanal läggas till eller tas bort dynamiskt.

2.9 Decentraliserad resursallokering

Mathiason och Amirijoo (2004) föreslår en decentraliserad lösning för resursallokering som kallas för *Real-time Communication Through a Distributed Resource Reservation Approach* (STRUTS). STRUTS använder sig av realtidskanaler som begränsar nätverkstrafiken genom att ta ett gemensamt beslut mellan de noder som påverkas av begränsningen. Realtidskanalerna är också skilda med avseende på hur viktiga de är (viktighetsklasser). För att klargöra hur allokering av kanaler i STRUTS går till visas detta med hjälp av ett kort exempel. Det finns tre olika viktighetsklasser i systemet som betecknas i_0, i_1 och i_2 där i_0 är den viktigaste av de tre. En kanal med viktighetsklass i_0 kan begränsa eller avallokera alla de existerande kanaler som är mindre viktiga, dvs. i_1 och i_2. Om de existerande kanalerna upptar den maximala bandbredden kommer en allokering av en kanal med viktighetsklass i_2 aldrig att kunna allokeras förrän någon kanal avallokeras eftersom i_2 är den minst viktiga av de tre viktighetsklasserna. Den kan heller inte avallokera eller begränsa existerande kanaler med samma viktighetsklass. Det gemensamma beslutet vid skapande av en kanal använder sig av *two-phase commit*

(2PC) (Gray, 1978) för att kunna garantera att den maximala kapaciteten inte kommer att överskridas på någon kommunikationslänk. Det finns dock vissa problem med 2PC som att det bygger på att nätverket inte får vara partitionerat samt att alla noder kommer vara låsta under processen då det gemensamma beslutet tas, dvs. de kan inte lägga till eller ta bort andra kanaler. Detta leder även till att andra noder under den tiden inte kan allokera kanaler för samma nod samtidigt, dvs. allokering är inte deterministisk. Om nätverket på något sätt skulle bli partitionerat kommer de noder som inte kan avgöra vad för beslut som tagits att vänta en godtyckligt lång tid tills ett beslut har bekräftats.

2.10 Schemaläggning i överbelastade realtidsdatabassystem

Schemaläggningen i överbelastade realtidsdatabassystem hanterar transaktioner. I realtidsdatabaser så behöver man en transaktionschemaläggare som ser till att transaktionerna möter sina deadlines. Schemaläggaren i DeeDS hanterar överbelastningar genom att abortera icke kritiska transaktioner. När dessa transaktioner involverar flera olika noder är det också viktigt att kommunikationen i hårda realtidssystem schemaläggs på sådant sätt att exempelvis viktiga transaktioner för realtidsdatabasen tillåts access till kommunikationslänken före mindre viktiga transaktioner. I denna rapport schemaläggs inte kommunikationen efter transaktioners kritikalitet, dvs. om transaktionen är hård eller mjuk, utan efter vad för den distribuerade realtidsdatabasen har för krav på replikeringen (sektion 2.5.1) mellan olika segment (sektion 2.5.2). Nedan beskrivs en algoritm för att schemalägga transaktioner på en processor. Denna algoritm kommer senare i denna rapport att anpassas för att istället schemalägga kanaler över ett nätverk.

Hansson m.fl. (1998) introducerar en ny schemaläggningsarkitektur med en algoritm för att lösa tillfälliga överbelastningar i en realtidsdatabas. Denna algoritm genererar en kostnadseffektiv plan för att lösa överbelastningar genom att avallokera schemalagda transaktioner så att nya transaktioner med högre prioritet får plats.

Hansson m.fl. (1998) skiljer mellan två typer av transaktioner *hard critical* och *firm*. I denna rapport används orden hård och mjuk för dessa två typer. Sett från ett realtidssystems perspektiv är det skillnad mellan firm och mjuk men detta bortses ifrån i denna rapport.

Det finns olika transaktionsklasser som existerar i systemet som urskiljs med hjälp av dess kritikalitet (Hansson m.fl., 1998). Kritikaliteten avgör om transaktionen har en hård deadline eller en mjuk deadline. Hårda transaktioners deadlines får ej brytas eftersom det kan leda till katastrof. Mjuka deadlines får brytas då systemet t.ex är överbelastat. Hårda transaktioner har en ersättningstransaktion med kortare exekveringstid som kan köras då system är överbelastat. Systemets arbetslast består av en mängd av transaktioner, $T = \{t_1, t_2, ..., t_n\}$, där varje transaktions kritikalitet är hård eller mjuk. Det finns också icke kritiska transaktioner och dessa får tas bort vid överbelastning så endast kritiska transaktioner exekveras.

Algoritmen som Hansson m.fl. (1998) har tagit fram försöker maximera summan av alla slutförda transaktioners utility, ett mått på hur viktigt en transaktion är i förhållande till andra transaktioner. Som tidigare nämnts körs endast ersättningstransaktioner då systemet är överbelastat. Dessa transaktioner tar generellt betydligt mindre resurser och kräver mindre tid vid exekvering jämfört med originaltransaktionen. Ersättningstransaktionen fungerar som en ersättare för originaltransaktionen som dock ger lägre utility. Det är då intressant att kunna minimera antalet ersättningstransaktioner

i systemet.

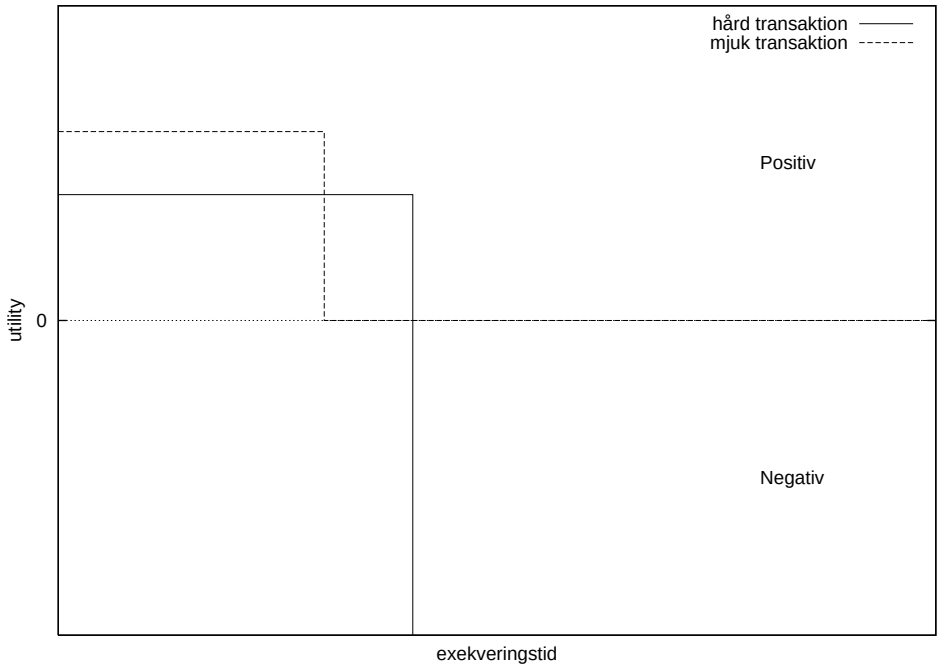
$$v_i(t) = \begin{cases} u_i > 0 & t \leq d_i \\ -\infty & k_i = \text{hard} \wedge t > d_i \\ 0 & k_i = \text{firm} \wedge t > d_i \end{cases}$$

Ekvation 1: Värdefunktionen (Hansson m.fl., 1998).

Ekvation 1 visar värdefunktionen som används. Värdefunktionen är hur stor önskan är att slutföra originaltransaktioner jämfört med deras ersättningstransaktioner. Dessa värdefunktioners resultat är hur mycket utility systemet kommer att få som en funktion av tiden det tar att slutföra transaktionen.

Det index i som används i ekvation 1 betecknar vilken transaktion den tillhör i mängden T som tidigare nämnts. Nedan förklaras vad de attribut som finns med i värdefunktionen betyder:

- d_i - transaktionens deadline.
- u_i - transaktionens utility.
- t - tiden från att transaktionen anlänt.
- k_i - transaktionens kritikalitet (hård eller mjuk).



Figur 6: Exempel på värdefunktioner för hårda och mjuka transaktioner.

Figur 6 visar exempel på hur värdefunktioner kan se ut. Den hårda transaktionen kommer att gå mot oändligheten när dess deadline bryts medan den mjuka transaktionen går till noll när dess deadline bryts.

2.10.1 Överbelastningsmodellen

Det som är viktigast att lösa är då överbelastning potentiellt kan göra att kritiska deadlines bryts (Hansson m.fl., 1998). För att kunna förhindra detta måste överbelastningsmekanismen vara känslig för vilken typ av överbelastning det är, dvs. vilken kritikalitet som håller på att brytas.

Då det existerar någon typ av överbelastning av systemet finns det två olika typer av handlingstyper som används i kombination för att lösa överbelastningen:

- Borttagning av icke kritiska transaktioner på ett kontrollerat sätt, dvs. göra ett urval och ta bort så lite som möjligt istället för att ta bort alla.
- Användning av ersättningstransaktioner som ersättning för originaltransaktionen.

Genom att välja vilka icke kritiska transaktioner som ska tas bort förbättras strävan efter att hålla en så hög utility som möjligt i systemet jämfört med om alla icke kritiska transaktioner skulle tas bort oavsett om det skulle räcka att bara ta bort några.

2.10.2 Arkitekturen

Arkitekturen består av fyra delar och dessa beskrivs kort nedan:

- **Dynamisk antagningskontroller (admission control)**
  Testar schemaläggningsbarheten, dvs. om det är möjligt att schemalägga den nya transaktionen med hänsyn på vad som tidigare givits tillåtelse att köra. Ser till så att schemaläggaren kan schemalägga transaktionen. Om det ej är möjligt att tillåta transaktionen direkt kommer överbelastningslösare att anropas för omallokering av resurser.
- **Schemaläggare av transaktioner (transaction scheduler)**
  Schemalägger den nya accepterade transaktionen med Earliest Deadline First (EDF) (Liu och Layland, 1973) och Stack Resource Policy (SRP) (Baker, 1991).
- **Överbelastningslösare (overload resolver)**
  Skapar en *overload resolution plan* (ORP) och utvärderar dess kostnads-effektivitet om det är lämpligt.
- **Exekveraren (dispatcher)**
  Utför exekvering enligt schemaläggningen som är bestämd av schemaläggaren av transaktioner.

2.10.3 Hantering vid överbelastning

Hanteringen av överbelastningen handlar om att ta fram en plan som löser överbelastningen genom att ta bort transaktioner som redan blivit accepterade av den dynamiska antagningskontrollern.

En handling som är en del av den plan som ska tas fram, som innefattar borttagning av icke kritisk transaktion eller ersättning av en kritisk transaktion, kallas för *overload resolution action* (ORA). Den plan som tas fram består av en mängd ORAs och kallas för *overload resolution plan* (ORP).

För att överbelastningslösaren ska kunna komma fram till vilka transaktioner som bör tas bort eller ersättas för att minimera förlusten av utility krävs tre typer av beräkningar:

- **Beräkning av sparad tid av ORA**
  Beräknar hur mycket tid som sparas genom användning av den specifika ORAn. Den sparade tiden av en specifik ORA kommer att bli mindre desto längre tid en transaktion har fått exekvera. Exempelvis om en transaktion kan tas bort helt är den sparade tiden hur mycket tid som är kvar för transaktionen att exekvera samt att den tar hänsyn till annat som t.ex. hur lång tid det tar för att ta verkligen ta bort själva transaktionen. Om en hård originaltransaktion istället skulle ersättas med en ersättningstransaktion kommer den sparade tiden att vara skillnaden mellan den längsta kvarvarande exekveringstiden och den längsta exekveringstiden för ersättningstransaktionen.

- **Beräkning av utility loss av ORA**
  Beräknar hur mycket utility som går förlorad då en originaltransaktion tas bort eller ersätts av en ersättningstransaktion. Vid borttagning kommer en visst tidsstraff att läggas till medans vid ersättning är den förlorade utility skillnaden mellan originaltransaktionen och ersättningstransaktionen.

- **Beräkning av utility loss density av ORA**
  Beräknar densiteten på hur stor förlusten är genom att ta utility loss delat med den sparade tiden. Denna beräkning används för att kunna välja en delmängd av möjliga ORAs eftersom detta är ett NP-svårt optimeringsproblem.

Överbelastningsalgoritmen som beskrivs i fyra steg nedan använder sig av de tre beräkningar som förklarades ovan för att göra det möjligt att välja ut den bästa mängden med ORAs som ger dem minimala sparade tiden för att få plats med den nya transaktionen. Algoritmen består av följande steg:

> "1. Generate the set of possible ORAs actions and compute their utility loss density.
>
> 2. Iterate through the ORAs in order of decreasing utility loss density, and add them to the ORP as long as the amount of saved time by the new ORP (overload resolution plan) does not exeed the time required.
>
> 3. From remaining actions, add the action with minimum utility loss, such that the required time is met or exceeded, to the set of selected actions.
>
> 4. Remove any now unnecessary actions from the set of selected actions. Start with the ORA with the highest utility loss, such that the required time is still met or exceeded." (Hansson m.fl., 1998)

Genom att använda utility loss density blir den sparade tiden kostnadseffektiv i jämförelse med andra ORAs.

3 Problembeskrivning

I dagsläget har DeeDS replikering inga garantier på att en uppdatering som ska ske på samtliga noder inom ett segment kommer att ske inom en förutsägbar tid, eftersom inget realtidsprotokoll används. De segment som finns i DeeDS har olika egenskaper, en av dessa egenskaper är vilken konsistensklass den tillhör. Av de konsistensklasser som nämns i sektion 2.5.1 finns det *bounded time* som innebär att replikeringen av data måste ske inom en förutsägbar tid. Det är då viktigt att kommunikationen mellan noderna sker med realtidskommunikation. Segment som förändras dynamiskt kräver också att schemaläggningen av kommunikationen mellan noderna är dynamisk och att systemet tolererar att noder kan krascha.

Ethernet är inte byggd för att hantera realtidskommunikation då protokoll som CSMA/CD används (Burns och Wellings, 2001). Detta gör att data inte kan skickas till en annan nod inom en förutsägbar tid. Under Ethernets utveckling har överföringshastigheten samt tillförlitligheten ökat. Detta har medfört att Ethernet har integrerats med system inom industrin (Martinsson, 2002). Datakommunikationen inom industin sker vanligtvis via någon typ av fältbuss och dessa karaktäriseras av hög tillförlitlighet, och låg överföringshastighet men är förhållandevis dyra. Två exempel på fältbussar som används inom automationen inom industrin är PROFIBUS (*PROFIBUS*, 2005) och CAN (ISO-11898, 1993). Genom Ethernets ökade tillförlitlighet och dess kostnadseffektivitet är det intressant att undersöka om det är möjligt att kunna tillhandahålla realtidskommunikation som kan ersätta de fältbussar som används inom industrin idag.

ThrottleNet är en lösning som kan få ett switchat Ethernet att skicka data inom en förutsägbar tid (Blomdell m.fl., 2004). Det möjliggörs genom att kontrollera att nätverkstrafiken på en kommunikationslänk inte överskrider dess maximala kapacitet. Om detta skulle ske kommer data att fylla bufferten i switchen och medföra att en flödeskontroll som är ungefär lika dålig som CSMA/CD börjar att användas, som i sin tur medför att kommunikationen ej längre blir deterministisk. Den existerande lösningen för dynamisk allokering av kanaler använder sig av en central nod, GlobeThrottle, som ser till att den maximala kapaciteten för en kommunikationslänk ej överskrids genom att bestämma vad som får allokeras. Att ha en central nod gör systemet sårbart och är något som helst bör undvikas när det övriga systemet är decentraliserat. Om denna nod skulle krascha kommer den dynamiska allokeringen att sluta fungera helt vilket bidrar till att hela systemet slutar att fungera. Vid användning av den decentraliserade lösningen förekommer ingen dynamisk allokering då endast statiska kanaler skapas vid start av systemet. Eftersom många system kräver att nätverket är dynamiskt uppbyggt där nya noder kan tillkomma eller tas bort bör det även finnas möjlighet att dynamiskt kunna skapa och ta bort kanaler. En decentraliserad resursallokering kan lösa problemet med den centrala noden i ThrottleNet genom att sprida ut kontrollen över flera noder och samtidigt göra det möjligt att få en dynamisk resursallokering. Mathiason och Amirijoo (2004) använder sig av two-phase commit (2PC) (Gray, 1978) för att skapa den decentraliserade resursallokeringen. Detta medför dock att systemet blir oförutsägbart när nätverket blir partitionerat, se sektion 2.9.

Av de arbeten som gjorts tidigare inom området, som Martinsson (2002) och Blomdell m.fl. (2004) skrivit om, har det inte tidigare funnits någon lösning som tar hänsyn till både hur kritisk själva kanalen är samt att allokeringen av kanalen ska ske inom en förutsägbar tid. Mathiason och Amirijoo (2004) har tagit hänsyn till hur viktig en kanal

är men har inte någon förutsägbar allokeringstid. Hur viktig en kanal ska vara beror på hur viktig kommunikationen är som överförs via kanalen. Exempel på detta är kanalen för *fly-by-wire*-funktionaliteten i ett passagerarflyg som anses vara den mest viktiga, medan autopiloten är mindre viktig för systemet. Det som Mathiason och Amirijoo (2004) inte skiljer på är hård och mjuk kanal som anger om kanaler får förändras eller avallokeras, när kanaler som är mer viktiga ska allokeras och måste få plats. Någon optimering av resursutnyttjande finns inte heller mer än att bara acceptera de kanaler som är mer viktiga.

3.1 Mål

Utveckla ett protokoll som tillåter feltolerant dynamisk schemaläggning av kanaler med olika kritikalitet över ett switchat Ethernet.

- **Feltolerant schemaläggning**
  En nod ska när som helst kunna krascha eller förlora kontakten med andra noder utan att påverka andra noders schemaläggning.
- **Dynamisk resursallokering**
  En nod ska under körning av systemet kunna allokera eller avallokera kanaler efter dess behov.
- **Hantera överbelastning med hänsyn till kanalers kritikalitet**
  Vid överbelastning bör de allokerade kanalerna schemaläggas på ett sådant sätt att de mest kritiska kanalerna accepteras och de mindre kritiska blir begränsade eller avallokerade. Endast de mjuka allokeringarna får avallokeras då hårda allokeringar är kritiska för systemets fortsatta korrekta funktion.

3.2 Antagande

I denna rapport görs följande antagande:

- All trafik som protokollet genererar i denna rapport antas få plats utan att överbelasta någon kommunikationslänk i systemet.
- Alla hårda kanaler som ska allokeras måste kunna gå att schemalägga.
- Det protokoll som tas fram klarar inte att hantera mer än en switch som alla noder är kopplade till, detta är också antagande som i ThrottleNet som tidigare nämndes om i sektion 2.8.
- När en nod kraschar antas denna krascha som *fail-silent*, dvs. utan att generera arbiträra meddelande.
- En switch antas alltid veta vilken nod som är kopplad till vilken port på switchen. Detta antagandet görs eftersom när switchen inte hittar mottagarnoden i dess cache genereras ett broadcast som i denna rapport inte kan hanteras.
- En kommunikationslänk antas aldrig krascha eller generera fel. Detta antagande något som diskuteras kring i en viktig punkt i framtida arbete.
- En switch antas alltid fungera korrekt och aldrig krascha.

4 Metod

Insamlingen av informationen i detta arbete sker genom en litteraturstudie då den mesta informationen inom området kan hittas i böcker, artiklar och journaler etc. För att informationen ska anses som relevant i detta arbete krävs det att den uppfyller något/några av de uppställda kriterierna nedan:

- Relaterat arbete.
- Refererats av relaterade arbete.
- Relaterad teknik som används i relaterade arbete.
- Tänkbara tekniker/algoritmer som kan uppnå något mål i detta arbete.
- Ge bättre förståelse inom området.

För att också försäkra sig om att något liknande arbete inte gjorts tidigare görs även sökningar baserad på exempelvis nyckelord kopplade till arbetet. Dessa sökningar görs på sidor på Internet som är kända för att tillhandahålla databaser med vedertagna artiklar, journaler och annat material.

Det huvudsakliga syftet med litteraturstudien är att ta fram ett protokoll som uppnår det uppsatta målet för denna rapport. Litteraturstudien har ThrottleNet (2.8) som utgångspunkt och därifrån sedan vidare undersöka de arbeten som är relaterade för att se vad som tidigare gjorts och vilka problem som fortfarande finns. Ett av de tidigare arbetena som är starkt kopplat till ThrottleNet är STRUTS (2.9) som också är en viktig del i denna studie. Efter en litterär studie av ThrottleNet, STRUTS och dess relaterade arbete bör två av delmålen kunna uppnås, dvs. *feltolerant schemaläggning* och *dynamisk resursallokering*.

Ett sidospår i litteraturstudien är hur överbelastning ska hanteras av protokollet, för att uppnå det tredje delmålet (hantera överbelastning med hänsyn till kanalers kritikalitet). Beroende på vad för typ av algoritm som ska användas i protokollet för hantering av överbelastningar ska det kunna passa in på krav som kommer från applikationer som använder sig av protokollet, exempelvis kan en sådan applikation vara DeeDS (2.5). De gränssnitt som också måste tas fram, skall ta hänsyn till vad som används för att uppnå delmålen.

En implementation görs som är baserad på de teorier som kommer fram av litteraturstudien. Detta för att kunna se om det rent praktiskt fungerar som det är tänkt samt att försöka se hur systemet påverkas av olika funktioner som används i systemet, till exempel hur hantering av överbelastningar (2.10) påverkar kommunikation som krävs vid upprättande av kanaler. Denna implementation ska inte vara en komplett modell av protokollet utan endast användas som vägledning för att möjligtvis hitta problem med protokollet som kan orsakas av olika val som görs.

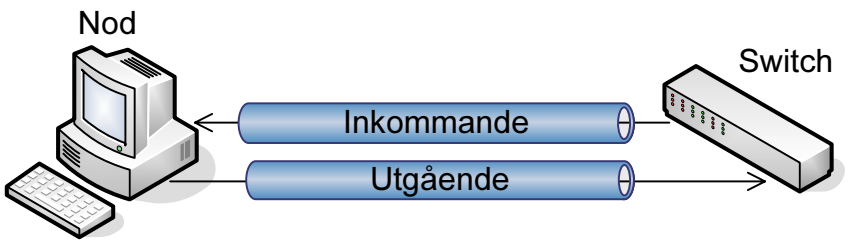
5 Distributed Resource Allocation Protocol

I denna sektion beskrivs protokollet Distributed Resource Allocation Protocol (DRAP) som är resultatet efter litteraturstudien som gjorts. Protokollet utvecklades för att uppfylla de mål som beskrivs i sektion 3.1 och för få bort vissa av de problem som beskrivs i sektion 2.9 som är knutna till 2PC. Exempel på detta är att noder är helt låsta under vissa delar av allokeringsprocessen vilket medför oförutsägbarhet om en nod kraschar.

DRAP är en vidareutveckling av decentraliserat ThrottleNet där dynamisk schemaläggning lagts till. Begränsningen av nodernas utgående trafik sker i TN-lagret som redan existerar i ThrottleNet (2.8).

5.1 Inkommande och utgående kanaler

En kanal är en logisk riktad ström som går från en nod till en annan nod via två kommunikationslänkar. Hos mottagarnoden kallar vi kanalen för inkommande och hos sändarnoden för utgående.



Figur 7: Kommunikationslänken uppdelad i inkommande och utgående.

Figur 7 visar kommunikationslänken mellan en nod och en switch som är uppdelad i en inkommande och en utgående kommunikationslänk. Kommunikationslänken kan samtidigt överföra data i båda riktningarna med maximal kapacitet (datamängd/tidsenhet). En kanal som går från en nod till en annan går via två kommunikationslänkar som tidigare nämnts. Dessa två kommunikationslänkar är anslutna till olika noder där båda är ansvariga för den kommunikationslänk den är ansluten till. En kanal tar upp en viss kapacitet av en inkommande eller utgående kommunikationslänk beroende på om det är en inkommande eller utgående kanal för noden som är ansluten till kommunikationslänken.

Allokering av inkommande kanaler begärs endast av andra noder, medan allokering av utgående kanaler begärs av noden själv. Samma princip gäller även vid avallokering eller begränsning av kanaler förutom då systemet är överbelastat. Överbelastning innebär exempelvis att summan av inkommande kanalers kapacitet överstiger den maximala inkommande kapaciteten som kan överföras via kommunikationslänken från switchen till noden. När systemet är överbelastat kan noden själv avallokera eller begränsa existerande inkommande kanaler som begärts av andra noder. Genom att ha denna uppdelning är det möjligt för andra noder att försöka göra en allokering av en inkommande kanal samtidigt som noden själv försöker göra en allokering av en utgående kanal.

5.2 Gränssnitt

DRAP består av två gränssnitt, ett externt och ett internt. Externa gränssnittet är det som applikationerna på noden använder sig av då den vill göra allokeringar eller avallokeringar av kanaler. Interna gränssnittet är det som noderna kommunicerar med mellan varandra för att kunna göra allokeringar och avallokeringar av kanaler.

5.2.1 Externt gränssnitt

I denna sektion presenteras det externa gränssnittet som applikationen kan använda sig av för att allokera eller avallokera kanaler.

- **allocateChannel(remoteNode, criticality, set<{capacity,utility}>)**

Funktionen allocateChannel körs då applikationen vill allokera en ny kanal. Följande argument anges:

  - *remoteNode*, anger vilken mottagarnoden är.
  - *criticality*, anger hur kritisk kanalen är för applikationen, dvs. om den är hård eller mjuk (beskrivs i sektion 5.8).
  - *set<{capacity,utility}>*, anger mängden möjliga alternativ en kanal kan ha.
    - *capacity*, anger kanalens kapacitet (datamängd/tidsenhet).
    - *utility*, anger hur värdefull kanalen är för systemet per tidsenhet (värde/tidsenhet) (beskrivs i sektion 5.8).

Denna funktion kommer att returnera kanalens id eller en felkod om allokeringen inte lyckades. Efter att en lyckad allokering av kanal används detta id för att kunna skicka meddelande ut på nätverket så att rätt kanal används.

- **deallocateChannel(id)**

Proceduren deallocateChannel avallokerar en kanal som är uppsatt mellan två noder. Följande argument anges:

  - *id*, kanalens id.

Efter att avallokering av kanalen har gjorts kommer trafik som försöker använda denna kanal inte att skickas ut på nätverket.

5.2.2 Internt gränssnitt

I denna sektion presenteras de olika meddelande som skickas mellan noderna.

- **Begäran om allokering av inkommande kanal / svar från mottagarnoden**

Meddelandet för begäran om allokering av inkommande kanal sänds till en annan nod när allocateChannel körs då applikationen vill allokera en inkommande kanal. Följande fält ingår i meddelandet:

  - *id*, kanalens id.
  - *criticality*, anger hur kritisk kanalen är för applikationen, dvs. om den är hård eller mjuk (beskrivs i sektion 5.8).

  - *set<{capacity,utility}>*, anger mängden möjliga alternativ en kanal kan ha.
    - *capacity*, anger kanalens kapacitet (datamängd/tidsenhet).
    - *utility*, anger hur värdefull kanalen är för systemet per tidsenhet (värde/tidsenhet) (beskrivs i sektion 5.8).

Detta meddelande är synkront i den bemärkelse att sändaren kommer att vänta tills att den har mottagit ett svar från mottagaren eller då ett fel inträffar som t.ex. att mottagaren har kraschat. Följande fält ingår i svaret från mottagaren:

  - *id*, kanalens id.
  - *{capacity,utility}*, valt par av kapacitet och utility.

- **Begäran om avallokering av inkommande kanal**

Meddelandet begäran om avallokering av inkommande kanal sänds till en annan nod när deallocateChannel körs då applikationen vill avallokera en kanal. Detta meddelande skickas även om det uppstår problem under allokeringsprocessen som t.ex. då det inte går att allokera kapaciteten lokalt men då det redan är allokerat hos mottagarnoden. Följande fält ingår i meddelandet:

  - *id*, kanalens id.

- **Begäran om avallokering av utgående kanal / bekräftelse från mottagarnoden**

Meddelande begäran om avallokering av utgående kanal sänds till påverkade noder då en mottagarnod har fått in en begäran om allokering av inkommande kanal som inte får plats utan att avallokera existerande kanaler. Följande fält ingår i meddelandet:

  - *id*, kanalens id.

Som optimering kan en bekräftelse användas som informerar sändaren om att begäran genomförts. Följande fält ingår i bekräftelsemeddelandet:

  - *id*, kanalens id.

- **Begäran om begränsning av inkommande kanal**

Meddelandet begäran om begränsning av inkommande kanal skickas till den påverkade noden under allokeringsprocessen. Meddelandet skickas då en allokering av en inkommande kanal lyckats hos mottagaren men en utgående kanal lyckades ej allokeras lokalt. Detta innebär att dess kapacitet måste sänkas för att kunna allokera kanalen. Följande fält ingår i meddelandet:

  - *id*, kanalens id.
  - *{capacity,utility}*, anger det som kanalen ska begränsas till.

- **Begäran om begränsning av utgående kanal / bekräftelse från mottagarnoden**

Meddelandet begäran om begränsning av utgående kanal sänds till påverkade noder då en mottagarnod fått in en begäran om allokering av inkommande kanal. Denna inkommande kanal som ska allokeras får inte plats och existerande kanaler måste begränsas. Följande fält ingår i meddelandet:

  – *id*, kanalens id.

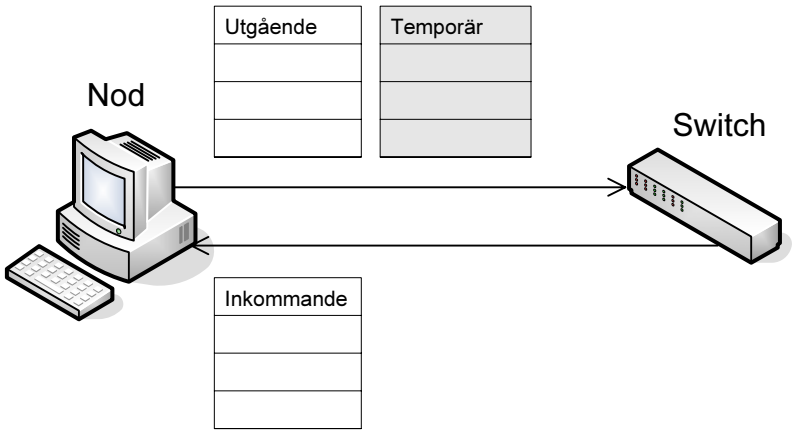
  – *{capacity,utility}*, anger det som kanalen ska begränsas till.

Som optimering kan en bekräftelse användas som informerar sändaren om att begäran genomförts. Följande fält ingår i bekräftelsemeddelandet:

  – *id*, kanalens id.

5.3 Allokeringslistor

För att ha kontroll på vilka allokerade kanaler som finns allokerade över kommunikationslänken kopplat till noden används tre listor. Dessa innehåller information om varje allokerad kanals kapacitet, utility och kritikalitet.

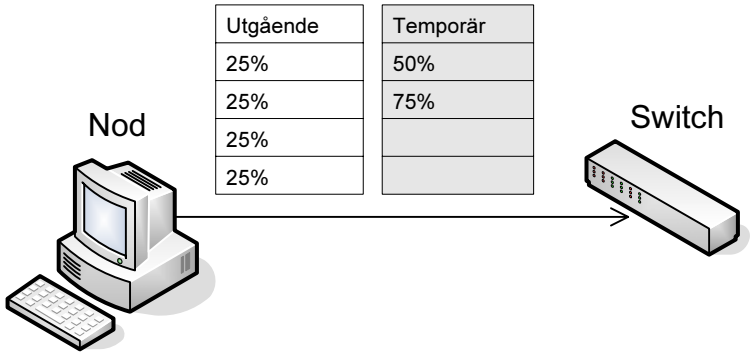


Figur 8: Listorna som används.

Dessa används för att ha kontroll på vad som är en pågående allokering samt vad som redan är allokerat: inkommande, utgående, temporär utgående (se figur 8). Den totala kapacitet som finns i listorna för inkommande och utgående kanaler får ej överskrida den maximala kapacitet som är möjlig att överföra via nodens kommunikationslänk i respektive riktning. Denna temporära lista får överstiga 100%, som senare kommer visas i ett exempel. För att lägga in en allokering av en kanal i den temporära listan krävs det att det finns tillräckligt med kapacitet kvar i utgående listan. Om detta inte finns är det möjligt att begränsa eller avallokera existerande kanaler för att få den nödvändiga kapaciteten. Viktigt att ta hänsyn till i detta läge är att de kanaler som ligger i den temporära listan också ska kunna begränsas eller avallokeras så att den nödvändiga kapaciteten kan nås.

Den temporära listan används för att förhindra onödiga begränsningar eller avallokeringar. Detta gäller då det är möjligt att allokera en utgående kanal på

sändarnoden men då det inte är möjligt att allokera en inkommande kanal på mottagarnoden. Utan den temporära listan skulle det innebära att sändarnoden kanske är tvungen att göra en del begränsningar eller avallokeringar av existerande kanaler innan beslutet från mottagarnoden är känt. Om då är fallet att mottagarnoden inte lyckas med att allokera en inkommande kanal innebär detta att sändarnoden måste avallokera sin utgående kanal helt. Detta innebär då att sändarnoden kanske har begränsat eller avallokerat andra kanaler i onödan för att få plats med den nya kanalen.



Figur 9: Exempel av relationen mellan utgående och temporär lista.

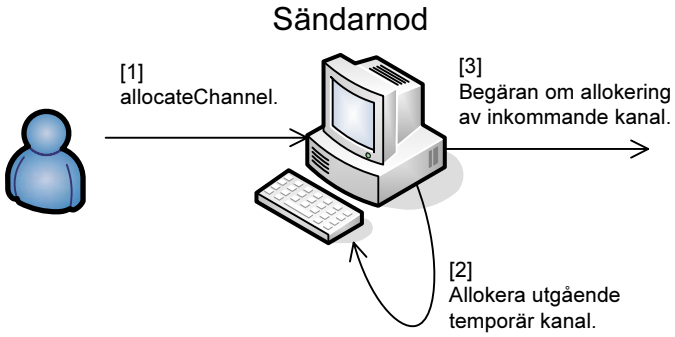
I följande exempel antar vi att den utgående listan innehåller fyra stycken allokerade kanaler som tar 25% av den totala kapaciteten var. Den temporära listan innehåller en pågående allokering av en kanal med kapaciteten 50% som kan ersätta två av de redan allokerade kanalerna. Om det vid detta tillfälle kommer in en ny begäran av utgående allokering som tar 75% kontrolleras denna mot både den temporära och den utgående lista och tittar på om summan av kapaciteten för de kanaler som den inte kan ersätta är tillräckligt liten så att den får plats. I detta fallet kommer den summan bli 25% eftersom adderas denna med den nya kapaciteten som ska allokeras på 75% kommer detta att gå eftersom kapaciteten inte överstiger 100%.

5.4 Allokeringsprocessen

I denna sektion beskrivs de tre faser som allokeringsprocessen kan delas upp i. Under varje fas beskrivs alla möjliga fall som kan hända under den fasen.

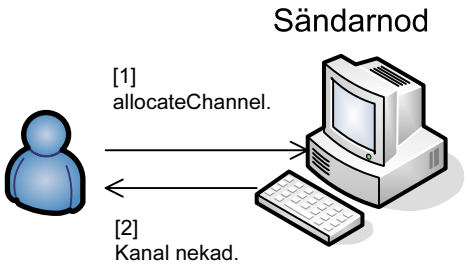
5.4.1 Fas 1

Denna fas börjar med att applikationen begär allokering av en kanal. Det finns då två fall som kan inträffa, *lyckad temporär allokering av kanal* och *misslyckad temporär allokering av kanal*. Dessa fall beskrivs nedan.



Figur 10: Lyckad temporär allokering av kanal.

Figur 10 visar en lyckad temporär allokering av kanal. När en allokering av en kanal begärts av applikationen [1] kommer det göras en kontroll om allokering av kanalen är möjligt att fullfölja med de kanaler som redan ligger allokerade. Av de alternativ som applikationen skickar in väljs det bästa möjliga alternativet som går att allokera. Det valda alternativet allokeras därefter som en temporär utgående kanal [2]. Efter detta kommer det skickas en begäran om allokering av inkommande kanal till mottagarnoden [3] med det bästa alternativet som noden valde och alla alternativ med lägre kapacitet.

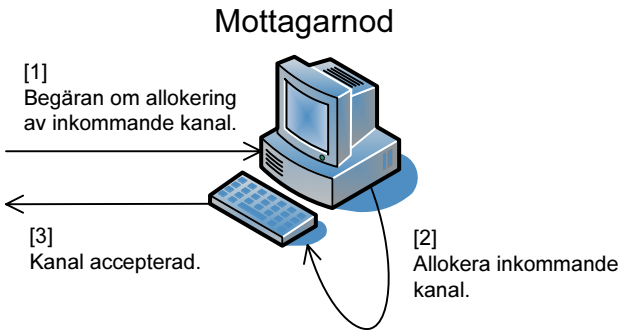


Figur 11: Misslyckad temporär allokering av kanal.

Figur 11 visar då kanalen inte är schemaläggningsbar (endast mjuka). Då kanalen ej är schemaläggningsbar kommer ingen temporär utgående kanal att allokeras och applikationen kommer direkt få ett svar tillbaka att den begärda allokering av kanalen blev nekad [2].

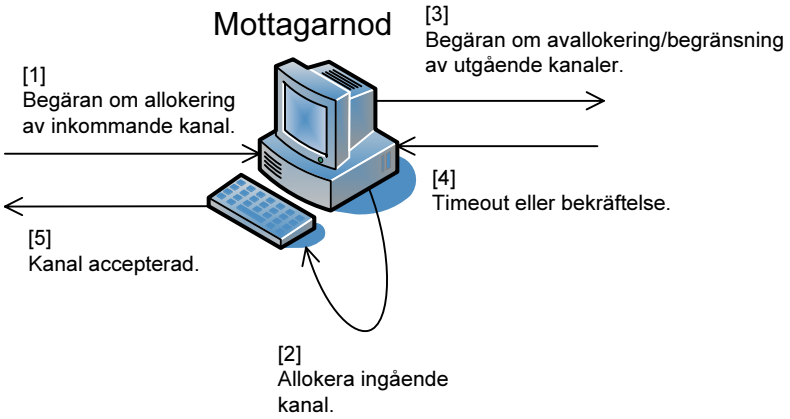
5.4.2 Fas 2

Givet att fas 1 lyckades skickas en begäran till mottagarnoden där det kan inträffa tre olika fall: *lyckad allokering av kanal utan avallokering*, *lyckad allokering av kanal med avallokering* och *misslyckad allokering av kanal*. Dessa fall beskrivs nedan.



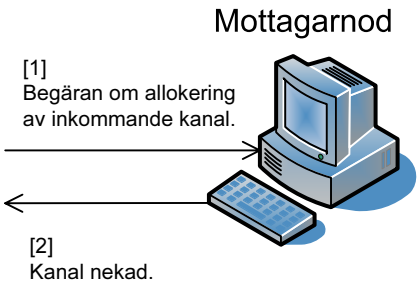
Figur 12: Lyckad allokering av kanal utan avallokeringar.

Figur 12 visar scenariot lyckad allokering av en kanal utan att behöva göra några avallokeringar av andra kanaler. Mottagarnoden tar emot begäran från sändarnoden [1] och kontrollerar först om allokering av kanalen är möjlig att schemalägga. Om det går att allokera utan några problem kommer den inkommande kanalen att allokeras [2] och sedan skickas ett meddelande tillbaka till sändarnoden att allokering av kanalen är accepterad och det alternativet noden valde [3].



Figur 13: Lyckad allokering av kanal med avallokeringar.

Figur 13 visar scenariot då redan existerande kanaler gör att det inte är möjligt att direkt allokera den inkommande kanal [1]. För att lösa en sådan konflikt kommer existerande kanaler att avallokeras eller begränsas, vilket beskrivs senare i sektion 5.8. Den inkommande kanalen allokeras [2] och därefter skickas begäran om begränsning eller avallokering av kanal till de noder som påverkas av begränsningen eller avallokeringen [3]. Efter att begäran om begränsning eller avallokering av kanal har skickats till alla påverkade noder väntar noden på en timeout eller tills den mottagit bekräftelse från samtliga av de påverkade noderna [4]. Efter detta kommer ett meddelande skickas tillbaka till sändarnoden om att allokering av kanalen accepterades [5].

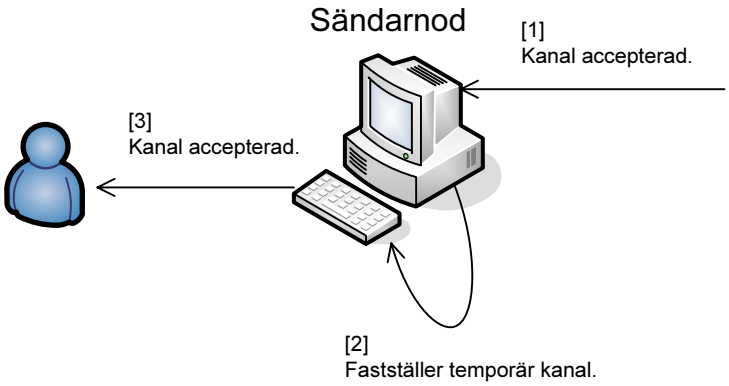


Figur 14: Misslyckad allokering av kanal.

Figur 14 visar scenariot då det ej är möjligt att allokera kanalen varmed ett meddelande skickas tillbaka till sändarnoden som säger att den inkommande kanalen nekades [2].

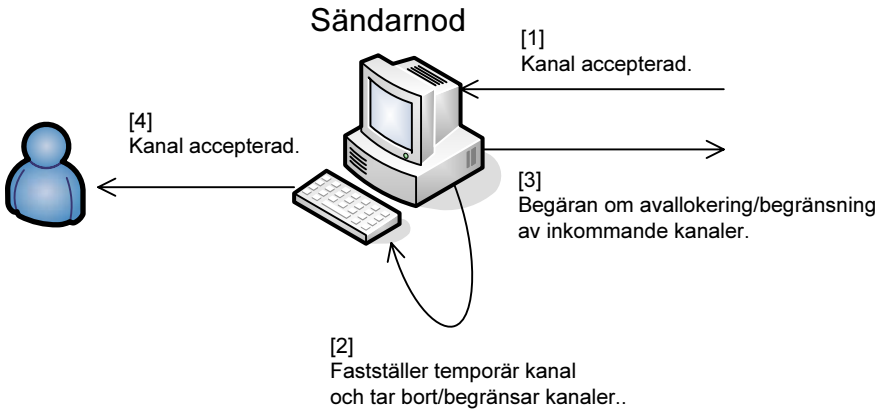
5.4.3 Fas 3

När sändarnoden får svar från mottagarnoden kan det inträffa fyra olika fall som beskrivs nedan: *lyckad allokering av kanal utan avallokering*, *lyckad allokering av kanal med avallokering*, *misslyckad allokering av kanal med avallokering* och *misslyckad allokering av kanal utan avallokering*.



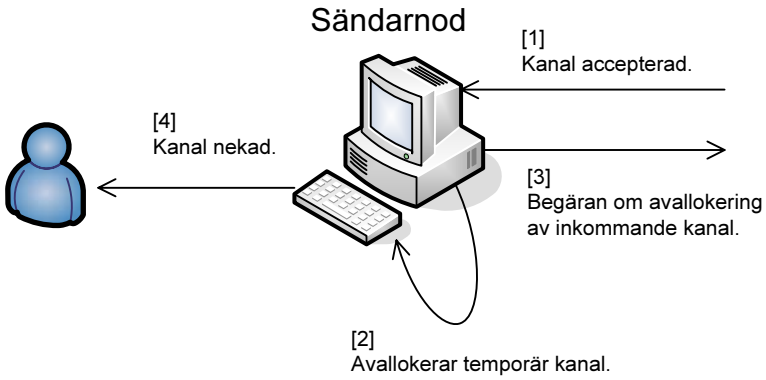
Figur 15: Lyckad allokering av kanal utan avallokeringar.

Figur 15 visar scenariot då mottagarnoden har accepterat kanalen [1] och sändarnoden fastställer den redan gjorda temporära kanalen genom att flytta över den temporära kanalen till listan för utgående kanaler [2]. Beroende på vilket alternativ mottagarnoden gjorde kommer noden att fastställa det bästa gemensamma alternativ. Efter detta är gjort meddelas applikationen att kanalen accepterades [3].



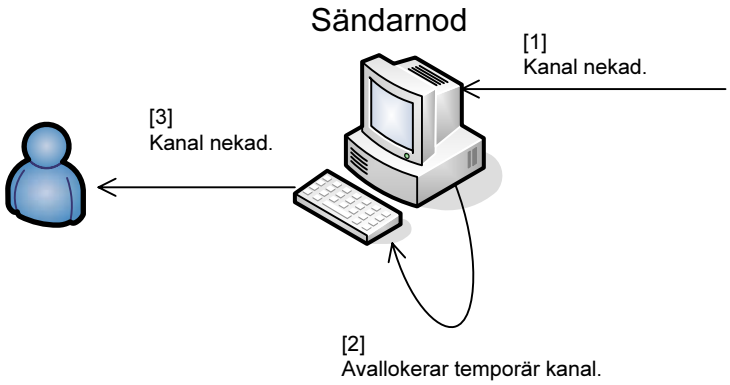
Figur 16: Lyckad allokering av kanal med avallokering.

Figur 16 visar då kanalen som accepterats av mottagarnoden [1] inte direkt kan allokeras utan andra kanaler måste begränsas eller avallokeras innan allokering av kanalen är möjlig. Innan begränsning eller avallokering utförs fastställs den nya kanalen [2] på samma sätt som tidigare nämnts men tar även bort eller begränsar de redan existerande kanalerna för att få tillräckligt med ledig kapacitet för att sedan skicka begäran om avallokering/begränsning av inkommande kanaler [3]. Om det bästa gemensamma alternativ som noden väljer är sämre än det som redan är allokerat på mottagarnoden kommer även ett begränsningsmeddelande att skickas till mottagarnoden. Detta scenario inträffar då någon annan kanal hunnit allokeras under tiden som sändarnoden väntade på svar från mottagarnoden gällande denna kanalen. Därefter meddelas applikationen att kanalen accepterats [4].



Figur 17: Misslyckad allokering av kanal med avallokering.

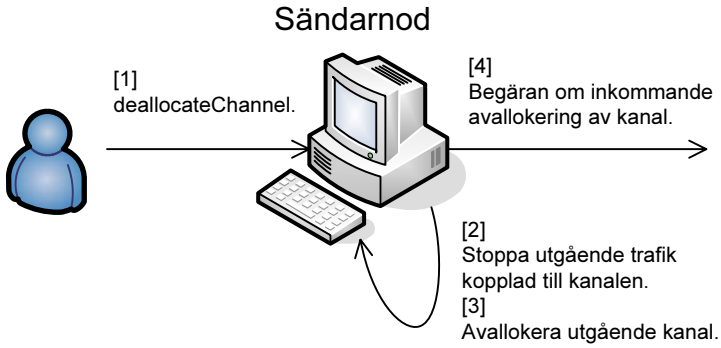
Figur 17 visar scenariot då andra utgående kanaler har fastställts innan denna nya kanalen som begärts fått svar från mottagarnoden [1] och medför att temporära kanalen inte kan fastställas då det inte är möjligt för den nya kanalen att avallokera andra kanaler för att få tillräckligt med kapacitet. När detta händer avallokeras den temporära kanalen [2] och sedan skicka en begäran ut om avallokering av den inkommande kanal som accepterades av mottagarnoden [3]. När detta är utfört kommer applikationen att meddelas om att kanalen nekades.



Figur 18: Misslycka allokering av kanal utan avallokering.

Om mottagarnoden nekat kanalen [1] som i figur 18 kommer den temporära kanalen att avallokeras [2] och applikationen kommer att meddelas att kanalen nekades [3].

5.5 Avallokeringsprocessen



Figur 19: Avallokering av kanal.

Figur 19 visar vad som händer när applikationen begär avallokering av kanal. Det första som händer är att det kommer ett anrop från applikationen som säger vilken kanal som ska avallokeras [1]. Användningen av den kanal som ska avallokeras kommer att stoppas [2] och sedan tas bort ur listan [3]. När detta är gjort är den utgående kanalen borta men den inkommande kanalen på mottagarnoden är inte avallokerad. Denna tas bort genom att ett meddelande skickas till mottagarnoden om begäran om avallokering av inkommande kanal [4]. Applikationen behöver ej vänta på svar från DRAP vid avallokering av kanal.

5.6 Hantering av begränsnings- och avallokeringsmeddelande

I sektion 5.4 (allokeringsprocessen) och sektion 5.5 (avallokeringsprocessen) beskrivs olika tillfällen under dessa processer då begränsnings- eller avallokeringsmeddelande skickas mellan noder. När en nod tar emot ett begränsningsmeddelande kommer noden att begränsa denna kanal till de värden som skickades med. När kanalen begränsats kommer noden inte att skicka något meddelande tillbaka till sändarnoden om det är en inkommande kanal. Om kanalen som begränsas är en utgående kanal kan en

bekräftelse skickas tillbaka till sändarnoden i optimeringssyfte. Detta innebär om sändarnoden får en bekräftelse att begränsningen är utförd av mottagarnoden innan sändarnodens timeout gått ut behöver inte sändarnoden vänta på att dess timeout ska gå ut. Avallokering kommer att ske på liknande sätt som vid begränsningen, fast istället för att begränsa kanalen kommer den att tas bort helt, och på samma sätt som vid begränsning kan bekräftelse skickas om det är en utgående kanal som påverkas.

5.7 Feltolerans

En typ av feltolerans som uppnås är att noderna när som helst ska kunna krascha utan att påverka schemaläggningen av kanaler mer än de kanaler som går från och till den kraschade noden (se sektion 2.2.1 för feltolerans i distribuerade system). För att uppnå denna typ av feltolerans låses inte noderna under allokerings- eller avallokeringsprocessen då kommunikation med inblandade noder sker för att undvika *hold & wait*. Denna typ av feltolerans bygger dock på att kommunikationen sker inom en förutsägbar tid för att exempelvis göra det möjligt att upptäcka fel, som att mottagarnoden har kraschat.

5.7.1 Avallokeringsprocessen

Vid avallokeringsprocessen, som beskrivs i sektion 5.5, kräver inte sändarnoden något svar från mottagarnoden vid avallokering av inkommande kanal. Om kontakten med mottagarnoden på något sätt skulle gå förlorad då exempelvis mottagarnoden har kraschat kommer sändarnoden inte vänta på att mottagarnoden eftersom en kraschad nod i denna rapport antas ha tappat alla sina allokerade kanaler. Eftersom deallocateChannel som tillhör det externa gränssnittet är ett asynkront anrop spelar det då ingen roll om mottagarnoden har kraschat eller ej eftersom denna information är onödig för applikationen att få reda på vid avallokering av en kanal.

5.7.2 Inkommande och utgående begränsning/avallokering

Under avallokeringsprocessen sker endast avallokering av inkommande kanaler hos mottagarnoden. Det finns även tre meddelanden till som skickas mellan noderna men som inte kontrolleras av applikationen som under avallokeringsprocessen. Dessa tre meddelande är: *begäran om begränsning av inkommande kanal*, *begäran om begränsning av utgående kanal* och *begäran om avallokering av utgående kanal*. Under avallokeringsprocessen väntar inte sändarnoden på något svar efter att begäran om inkommande avallokering av en kanal har skickats. Detta gäller även för begäran om inkommande begränsning. Skillnad i beteende kommer när de två kvarvarande meddelandena skickas, *begäran om begränsning av utgående kanal* och *begäran om avallokering av utgående kanal*. Eftersom det är en utgående kanal för den mottagarnoden som meddelandet skickas till är det viktigt att den noden verkligen har slutat att skicka eller begränsat kanalen innan den nya kanalen accepteras på sändarnoden eftersom annars finns det risk för att systemet blir överbelastat.

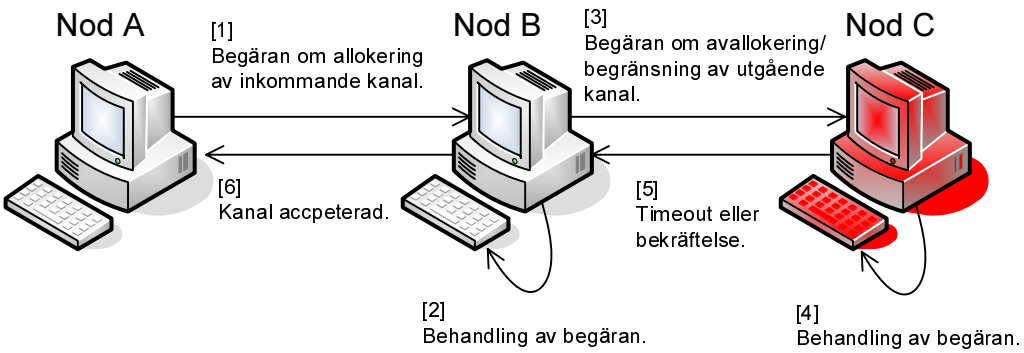
Dessa meddelande kan använda sig av ett bekräftelsemeddelande från mottagarnoden men ses endast som optimering i denna rapport, då liknande PAR-protokollet användas som beskrivs i sektion 2.4.1. Den timeout som sätts antingen vid användning av PAR eller bara vid användning av timeout sätts till den längsta tiden det kan ta, dvs. längsta tiden det tar att skicka begränsnings- eller avallokeringsmeddelandet till mottagarnoden samt tiden det tar att göra begränsningen eller avallokeringen av kanalen hos mottagarnoden. Efter att denna tid har gått ut antas mottagarnoden ha begränsat

eller tagit bort kanalen som skulle begränsas eller avallokeras. Oavsett hur många begränsningar eller avallokeringar av kanaler som sker samtidigt sätts denna timeout till denna längsta tid.

5.7.3 Allokeringsprocessen

Vid allokeringsprocessen, som beskrivs i sektion 5.4, kräver sändarnoden ett svar från mottagarnoden då begäran om allokering av utgående kanal har skickats. Givet att tiden för att skicka allokeringsmeddelandet är förutsägbart och mottagarnodens exekveringstid är förutsägbar kommer också längsta tiden det tar att få ett svar att vara känd. Denna tid består av tiden det tar att skicka allokeringsmeddelande, tiden det tar att göra allokeringen på mottagarnoden inklusive tiden det tar att göra eventuella avallokeringar, och tiden det tar för att skicka svaret tillbaka till sändarnoden. Själva processen kring att skicka allokeringsmeddelande till mottagarnoden sker på liknande sätt som i PAR-protokollet, se sektion 2.4.1, där timeouten sätts till den längsta tiden som nämndes tidigare. Om inget svar har kommit tillbaka innan timeouten har gått ut antas mottagarnoden ha förlorat alla inkommande och utgående kanaler och applikationen informeras om felet.

Ett exempel nedan beskriver hur allokeringsprocessen ser ut när begränsningar eller avallokeringar av andra kanaler måste ske på mottagarnoden för att den nya kanal ska kunna allokeras.



Figur 20: Exempel på när det är viktigt med användning av timeout för att få förutsägbara allokeringstider.

Figur 20 visar ett exempel på detta där nod A skickar en begäran om allokering av inkommande kanal [1] till nod B och där nod B är tvungen att begränsa eller avallokera redan existerande inkommande kanaler för att få plats med den nya kanalen [2]. Nod B skickar ut begäran om begränsning eller avallokering av utgående kanal [3] till nod C och väntar på att en timeout [5] ska gå ut som är satt till värsta fallet, dvs. tiden det tar att skicka avallokeringsmeddelandet till nod C [3] samt tiden det tar att göra avallokering av den utgående kanalen hos noden C [4]. Ett svar från nod C innan timeouten gått ut används endast för optimeringssyfte för slippa vänta på timeouten [5]. När timeouten gått ut antas nod C mottagit begäran om avallokering av utgående kanal och utfört den eller så har kontakten tappats varmed det antas att nod C förlorat allokeringen av kanalen. Efter detta skickas ett svar tillbaka till nod A att kanalen har accepterats [6].

5.8 Schemaläggning vid överbelastning

För att DRAP ska kunna hantera överbelastningar krävs det en algoritm som kan göra lämpliga urval på vad som ska behållas, förändras eller tas bort. En sådan algoritm beskrivs i sektion 2.10 som används vid kortvariga överbelastningar i ett realtidsdatabassystem. Att anpassa algoritmen för att fungera ihop med DRAP är inte helt trivialt eftersom DRAP schemalägger kanaler över ett nätverk medan algoritmen i sektion 2.10 schemalägger transaktioner på en processor, se sektion 2.7. Detta kommer att beskrivas under denna sektion för att klargöra skillnaderna mellan algoritmen som beskrivs i sektion 2.10 och denna.

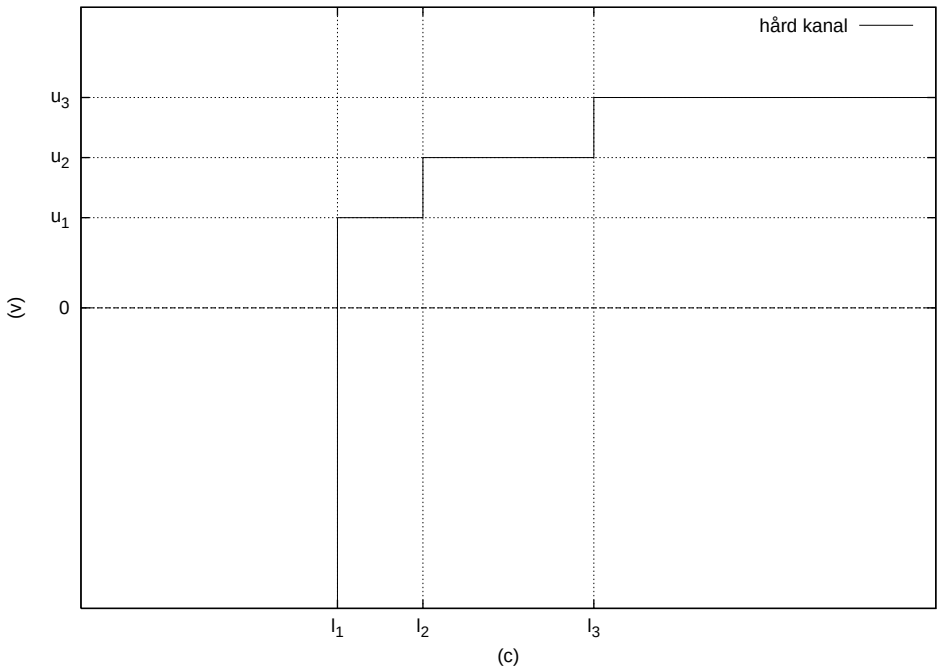
Kritikalitet är ett mått för hur kritisk en kanal är för applikationen eller användaren. Det finns två typer av kritikaliteter, hård och mjuk, som beskrivs i sektion 2.10. Kanaler som är hårda har flera alternativa kapaciteter som vid överbelastning används när kanalen begränsas medan mjuka kanaler inte har detta. Utility är ett mått som är kopplad till en kanal som anger hur värdefull den är för systemet per tidsenhet (värde/tidsenhet). Denna utility kan jämföras med utility i sektion 2.10 men det som skiljer är att utility i sektion 2.10 inte är per tidsenhet. För att tydliggöra skillnaderna mellan dessa två mått i DRAP och algoritmen som beskrivs i Hansson m.fl. (1998) förklaras detta i texten som följer.

5.8.1 Värdefunktion

Den värdefunktion som beskrivs kort om i sektion 2.10 skiljer sig med den typ av värdefunktion som krävs för att göra det möjligt att fungera ihop med schemaläggning vid av kanaler vid överbelastning.

$$v(c) = \begin{cases} u_1 > 0 & l_2 > c \geq l_1 \\ u_2 > u_1 & l_3 > c \geq l_2 \\ \vdots & \vdots \\ u_n > u_{n-1} & c \geq l_n \\ -\infty & \text{hård} \wedge c < l_1 \\ 0 & \text{mjuk} \wedge c < l_1 \end{cases}$$

Ekvation 2: Värdefunktionen.

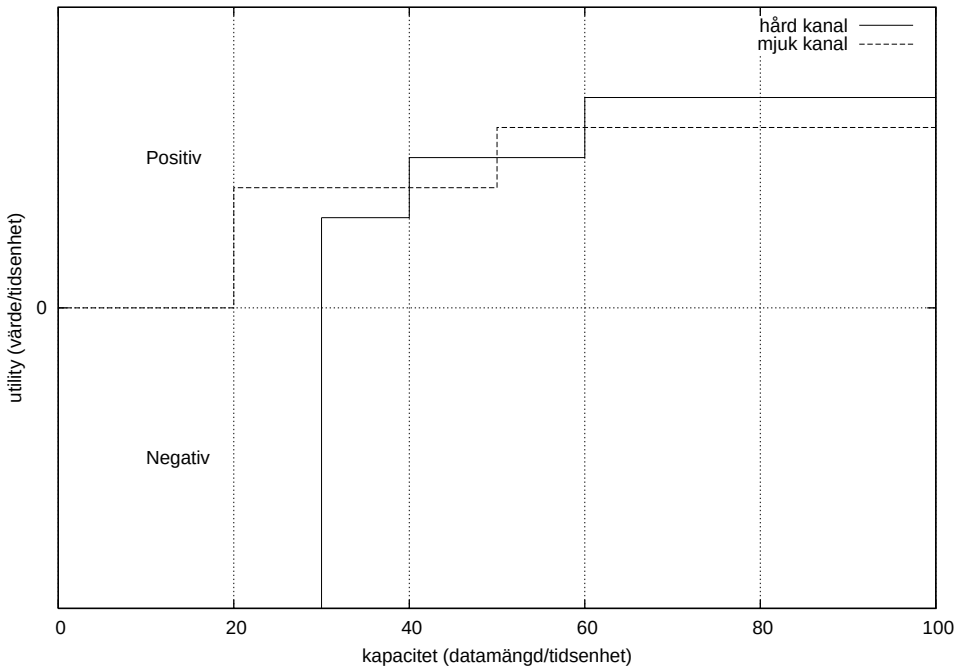


Figur 21: Värdefunktionens graf.

Figur 21 visar värdefunktionen (v) som används i DRAP och kan jämföras med värdefunktionen i sektion 2.10 ekvation 1. Skillnaden är att värdefunktionen i ekvation 2 inte har några separata värdefunktioner för de hårda kanalerna som kan begränsas med olika alternativ, utan allt ligger i samma värdefunktion. Indexet n i ekvation 2 står för antalet alternativ en kanal har. Figur 21 visar ett exempel på hur de olika variablerna i värdefunktionen i ekvation 2 är kopplade till varandra.

Nedan förklaras vad de attribut som finns med i värdefunktionen betyder:

- l_1 - kanalens absolut lägsta kapacitet (datamängd/tidsenhet).
- $l_{1...n}$ - kanalens lägsta kapacitet vid olika alternativ (datamängd/tidsenhet).
- $u_{1...n}$ - kanalens utility vid olika alternativ (värde/tidsenhet).
- c - kanalens kapacitet (datamängd/tidsenhet).



Figur 22: Exempel på värdefunktioner för hårda och mjuka kanaler.

Figur 22 visar exempel på hur resultatet av värdefunktioner för hårda och mjuka kanaler kan se ut i den omgjorda algoritmen som används i DRAP. För att beskriva varför figuren ser ut som den gör kommer ett exempel med videoströmmar att beskrivas. En video ska strömmas från en nod till en annan med tre olika kvalitetsnivåer, vilket kan jämföras med de tre steg som linjen för den hårda kanalen har i figur 22. Den bästa kvaliteten kräver 60% av den totala bandbredden, kapaciteten (datamängd/tidsenhet), som går att skicka med, och den sämsta kräver 30%. Om nu bandbredden inte skulle räcka till för att se på videoströmmen med den lägsta kvaliteten kommer filmen att börja hacka och medför att den som tittar på den tycker det inte går att titta på längre. Hur den som tittar på videoströmmen upplever kvaliteten kan jämföras med hur bra utility (värde/tidsenhet) en viss kapacitet ger systemet. Den mjuka kanalen i figur 22 kan jämföras med en videoström som har två nivåer och där videoströmmen inte är så viktig och kan därmed få avbrytas vid brist av tillräcklig bandbredd.

5.8.2 Beräkning av sparad kapacitet

I sektion 2.10 beskrivs den sparade tiden som är kopplad till en ORA. Hur mycket tid som sparas beror på hur länge den transaktionen som ska ersättas har kört. Eftersom den modifierade algoritmen som används i DRAP inte använder sig av deadlines kommer det inte heller vara möjligt att beräkna sparad tid utan denna ersätts av sparad kapacitet (datamängd/tidsenhet). Till skillnad från sparad tid kommer sparad kapacitet inte att förändras beroende på hur lång tid det har gått utan kommer vara konstant.

$$\xi = c$$

Ekvation 3: Beräkning av sparad kapacitet vid borttagning.

Beräkningen av den sparade kapaciteten (ξ) vid borttagning av mjuka kanal visas i ekvation 3 där c är kapaciteten för kanalen som ska tas bort.

$$\xi = c - d$$

Ekvation 4: Beräkning av sparad kapacitet vid begränsning.

Den sparade tiden, beskrivs i sektion 2.10.3, förändras på olika sätt beroende på vad för kritikalitet den har eftersom när hårda transaktioner ersätts är den sparade tiden skillnaden mellan den nya transaktionen och ersättningstransaktionen. Skillnaden är att vid ersättning av en originaltransaktion minskas den sparade tiden med ersättningstransaktionens längsta exekveringstid. Denna typ av skillnad finns inte i den algoritm som används i DRAP, eftersom det inte är sparad tid som beräknas utan sparad kapacitet. För att beräkna den sparade kapaciteten då en kanal ska begränsas (ersättas) är d i ekvation 4 den alternativa kapacitet kanalen ska begränsas till och c är kanalens nuvarande allokerade kapacitet. Det kanalen begränsas till är ett av de alternativa kapaciteter som skickades med då applikationen först begärde allokering av kanalen.

5.8.3 Beräkning av utility loss

Den beräkning som görs av utility loss för en specifik ORA i Hansson m.fl. (1998) är likvärdig den beräkning för att beräkna fram utility loss i DRAP. Skillnaderna mellan dessa är att i DRAP är utility loss per tidsenhet samt att beräkningen av utility loss inte påverkas av tiden utan av hur stor kapacitet kanalen har.

$$\gamma = v(c) - v(d)$$

Ekvation 5: Beräkning av utility loss.

Ekvation 5 visar beräkningen av utility loss (γ) i DRAP. Vid borttagning av en kanal är c i figuren ovan större eller lika med kanalens absolut lägsta kapacitet och d är lika med noll. Detta innebär att $v(c)$ ger den utility kanalen har vid dess nuvarande satta kapacitet och $v(0)$ är ett straff som läggs till för att kanalen ska tas bort. Detta straff varierar beroende på vad för kritikalitet kanalen har. Om kanalen är mjuk är straffet noll och om kanalen är hård är straffet oändligt eftersom hårda kanaler inte får tas bort.

Begränsning av en kanal beräknas på liknande sätt som i ekvation 5 men istället för ett straff kommer den utility som ges vid ett begränsat alternativ att dras bort, skillnaden blir förlusten av utility. I ekvation 5 kommer d att vara lika med den alternativa kapacitet som kanalen ska begränsas till som är större eller lika med kanalens absolut lägsta kapacitet. Detta innebär att systemet alltid kommer att få en viss förlust av utility även vid begränsning av en kanal.

5.8.4 Beräkning av utility loss density

För att kunna avgöra om två ORAs är lika viktiga beräknas utility loss density som tar hänsyn till hur mycket resurser ORAn sparar.

$$\frac{\gamma}{\xi}$$

Ekvation 6: Beräkning av utility loss density.

Ekvation 6 visar beräkningen av utility loss density där utility loss (γ) divideras med den sparade kapaciteten (ξ) som beräknats. Denna utility loss density kommer vara konstant förhållande till tiden som gått. Till skillnad från utility loss density i Hansson m.fl. (1998) som stiger desto längre tid transaktionen har fått exekvera.

5.9 Förutsägbara allokerings- och avallokeringstider

Kommunikationen som genereras av DRAP antas vara förutsägbar i denna rapport. Detta kommer endast att diskuteras kort om vilka problem som finns och vad som är viktigt att tänka på för att uppnå detta antagande. För att åstadkomma förutsägbara allokerings- och avallokeringstider kan detta göras genom att vid start allokera statiska kanaler som bara används för denna typ av trafik. Frågan är hur mycket kapacitet som måste allokeras för att kunna hantera den trafik DRAP genererar på nätverket. Svårigheten är att kunna komma fram till en enkel formel som användaren av DRAP kan använda sig av. Exempelvis ska användaren bara behöva veta hur många noder som finns vid start och hur många allokeringar eller avallokeringar som ska kunna hanteras per tidsenhet. Den del av DRAP som påverkar trafiken mest är schemaläggningen av kanaler då systemet är överbelastat. Trafiken som genereras vid en överbelastning kan vara betydligt större jämfört med ett icke överbelastat system. Hur mycket trafik som genereras beror på existerande och nya kanalernas storlek, kritikalitet och utility. Anta exempelvis att den totala kapaciteten skulle vara helt utnyttjad och alla de existerande kanalerna i systemet bara skulle vara mjuka kanaler där varje kanal tar upp 1% av den totala kapaciteten. Beroende på kapaciteten för nästa kanal som måste allokeras, kommer trafik som genereras vid överbelastningen att bero på hur mycket kapacitet den nya kanalen tar. Om kanalen tar 1% eller lägre kommer minimalt med trafik att genereras men om den är högre än 1% kommer trafiken som genereras att öka ju större differensen blir mellan den nya kanalens kapacitet och den kapacitet som varje existerande kanal har, dvs. 1%. Detta innebär att den nya kanalen måste ha en minimal kapacitet för att begränsa antalet avallokeringar.

6 Diskussion

Målet i denna rapport är att skapa en feltolerant distribuerad protokoll, DRAP, för att allokera och schemalägga kanaler med olika kritikalitet på ett switchat Ethernet. Den förutsägbara resursallokeringen uppnås i DRAP genom att undvika att låsa noder under tiden kanaler håller på att allokeras. Överenskommelsen mellan noderna vid allokering av en kanal sker inte atomärt, dvs. allokering av kanalen fastställs på mottagarnoden innan sändarnoden, men för applikationen som använder DRAP sker det atomärt. Genom att förhindra att noder låses vid allokering av kanaler förhindras *hold & wait* vilket underlättar felhanteringen då noder kraschar. Att noder inte låses förhindrar även att schemaläggning av kanaler inte störs då noder kraschar mer än till den kraschade noden. Schemaläggning i DRAP är baserad på Hansson m.fl. (1998) men är anpassad för att kunna hantera tillfälliga överbelastningar på en kommunikationslänk.

Ett exempel på en applikation som kan använda sig av DRAP är DeeDS (2.5) för att uppnå förutsägbar replikering av data på ett switchat Ethernet. I DeeDS existerar det något som kallas för segment (2.5.2) som har olika egenskaper. En sådan egenskap kan ange exempelvis om data ska replikeras inom en förutsägbar tid. Detta är något som då kan uppnås vid användning av DRAP eftersom DeeDS idag använder sig av switchat Ethernet.

Andra typer av användningsområden som DRAP skulle kunna vara aktuell för är till exempel olika typer av simulatorer som är sammankopplade via ett nätverk. Simulatorerna kan i detta fall vara krigsimulatorer som simulerar olika typer av enheter som exempelvis pansarvagnar eller flyg. DRAP göra det möjligt att garantera bandbredd mellan simulatorerna via ett switchat Ethernet då trafiken även anses ha olika kritikalitet. Exempel på detta kan vara då replikering av enheters positioner sker inom en förutsägbar tid samtidigt som annan mindre kritisk informations skickas via nätverket.

6.1 Relaterade arbete

RETHER är en lösning som använder COTS-switchar och endast gör förändringar i de existerande noderna (Chiueh och Venkatramani, 1994). Används istället för CSMA/CD när realtidstrafik skickas på nätverket. Vid realtidstrafik aktiveras RETHER och ett *token-passing*-protokoll används, där en *token* skapas som sedan cirkulerar mellan noderna. Användandet av en *token* som cirkulerar medför problem exempelvis då *token* går förlorad jämfört med att använda DRAP som inte bygger på ett *token-passing*-protokoll. För att öka feltolerans har Chiueh och Venkatramani (1997) gjort att den noden som haft *token* senast blir övervakare för efterföljande nod. Detta gör att det krävs att båda dessa noder måste krascha för att *token* ska försvinna eftersom den övervakande noden annars skapar en ny token om efterföljaren kraschar. RETHER använder sig av *token-passing*-protokoll som är tänkt för periodisk trafik som exempelvis videokonferenser. I DRAP kan både periodisk och sporadisk trafik hanteras. Detta är möjligt eftersom hur trafiken ser ut som skickas via en kanal inte har någon betydelse då kanaler begränsar hur mycket bandbredd som får användas. Någon schemaläggning görs heller inte utan *token* cirkulerar endast som *Round Robin* (RR). RETHER motverkar överbelastningar genom att en nod inte får allokera bandbredd om det inte finns tillräckligt med bandbredd ledigt. Det finns heller inte något som exempelvis tar hänsyn till hur kritisk realtidstrafiken från en nod är i jämförelse med realtidstrafik från en annan nod. Token skapas när RETHER aktiveras och cirkulerar

över två typer av noder, real-time (RT) och non-real-time (NRT) där noder som är av NRT endast får skicka data om det finns tid över. Denna typ av skillnad i DRAP existerar inte men kan istället använda mjuka kanaler för att kunna tillåta *best-effort*-trafik på nätverket. Venkatramani och Chiueh (1997) beskriver en utökning av RETHER som klarar av att hantera trafik i nätverk med flera switchar. DRAP har inte detta stöd och klarar bara av att endast en switch används eftersom den inte kan kontrollera hur mycket som flödar mellan två switchar och riskerar då överbelastning av kommunikationslänken.

Varadarajan och Chiueh (1998) beskriver EtherReal som inte kräver några förändringar av de existerande noderna utan endast förändringar i switcharna som noderna är kopplade till. Målet med EtherReal är att skapa en skalbar Ethernet-switch med realtidsegenskaper. Genom att göra dessa förändringar kan hård realtidstrafik uppnås men det fungerar inte att använda en COTS switch. Istället användas COTS-hårdvara som i prototypen var en dator med fyra nätverkskort där Linux används som operativsystem. Detta är något som troligtvis leder till en ökad kostnad jämfört med att använda sig av COTS-switchar. Realtidsapplikationen på någon nod skickar begäran om allokering av bandbredd till en process, *Real-Time Communication Daemon* (RTCD), på samma nod. Denna process har som ansvar att påbörja och avsluta kopplingar med realtidskrav. För att se till att realtidsapplikationerna håller den bandbredd som är allokerad existerar ytterligare en process som har detta som ansvar, *Real-Time data Transmission/Reception* (RTTR). Problemet med realtidsapplikationen, RTCD och RTTR är att processerna ligger i *user-space* och medför att trafik som genereras av operativsystemet som exempelvis ARP eller RARP inte kan kontrolleras men den jämnas dock ut i EtherReal switchen. I likhet med RETHER motverkar också EtherReal överbelastningar och medför att ingen hantering av överbelastningar behövs. EtherReal stödjer ingen eller endast begränsad hård realtidstrafik vilket är något som DRAP stödjer och har som krav att kunna stödja.

Hoang m.fl. (2002) presenterar en lösning där förändringar måste göras i både noderna och switchen men där ett lager läggs precis ovanför *link layer*. Detta gör att de kan garantera bithastighet och tidskrav för periodisk trafik. Även denna lösning bygger på att använda ett *token-passing*-protokoll, men även klocksynkronisering som är en del av switchens uppgift. Schemaläggningen av trafiken sker med hjälp av EDF och existerar i både noderna och switchen. Trafiken schemaläggs utefter relativa deadlines som är en del av den informationen som finns tillgänglig när kanalen sätts upp. Detta är något som DRAP inte schemalägger efter. Såsom RETHER har även denna lösning stöd för att kunna hantera både realtidstrafik och icke-realtidstrafik. I likhet med DRAP så använder sig även denna lösning av kanaler som är dynamiska där en förfrågan ner till ett RT-lagret görs när en applikation vill sätta upp en ny kanal för realtidstrafik.

Loeser och Härtig (2004) tar fram en lösning som visar att det går att få hård realtidskommunikation med låg fördröjning. En drivrutin kallad *RT-net* skapas för att kommunicera direkt med nätverkskortet som tar hand om all allokering av bandbredd för den data som ska skickas via ett switchat Ethernet. Vid allokering av bandbredd accepteras allokeringen först lokalt och skickas sedan vidare till en dedicerad nod som har kontroll på alla nuvarande kopplingar. Detta medför att det finns en nod i nätverket som kan förstöra allt om denna kraschar (*single-point-of-failure*) och det är en egenskap som inte är önskad i DRAP. Dessutom motverkas överbelastningar att ske vid allokering av bandbredd.

STRUTS (Mathiason och Amirijoo, 2004) som tidigare nämnts om i sektion 2.9 använder sig av 2PC för att komma överens mellan noderna då kanaler skapas. Vid pågående allokering av kanaler kommer de inblandade noderna att vara låsta vilket medför att andra allokeringsförsök under den tid inte kommer vara möjliga att göra. DRAP blockerar inte vid utgående allokering av kanal och medför att flera processer på samma nod inte blir blockerade under tiden andra processer försöker att allokera nya kanaler på olika mottagarnoder samtidigt. Om en pågående allokering av en kanal på något sätt inte lyckas genom att en nod som den behöver kontakta eller få svar ifrån har kraschat kommer STRUTS allt hålla kvar låsningen på de inblandade noderna tills något svar eller liknande har erhållits från den kraschade noden. Med DRAP kommer detta inte att ske, och de som påverkas är noder som har något att göra med den kraschade noden. För att DRAP ska kunna hantera fel på ett bättre sätt krävs det att det är byggt på förutsägbara nätverks- och exekveringstider då exempelvis timeouts kan sättas till den tid tar för svar från en nod ska ha kommit tillbaka. Detta innebär att den i princip kommer vara lika dålig som STRUTS i förutsägbarhet om inte förutsägbara allokerings- och avallokeringstider kan ges. En annan nackdel i DRAP är om en inkommande allokering av en kanal på en mottagarnod lyckas men inte lyckas lokalt kommer det innebära längre exekveringstid än om all allokering skulle ske efter ett gemensamt beslut som i STRUTS.

6.2 Framtida arbete

I denna sektion presenteras möjliga framtida arbete på det som gjorts samt att det diskuteras kort om vad som kan vara intressant att undersöka vidare på inom dessa arbeten.

6.2.1 Förutsägbara allokerings- och avallokeringstider

I denna rapport antas att tiden från då resursallokeringen begärs tills dess att kanalen har blivit accepterad eller ej ska vara förutsägbar. Det vore intressant att undersöka huruvida det är möjligt att komma fram till en formel för beräkning av hur stor datamängd/tidsenhet som krävs för allokerings- och avallokeringstrafiken. Detta resultat kan användas för att sätta upp statiska kanaler mellan noderna vid start för att få förutsägbara allokerings- och avallokeringstider.

6.2.2 Användning av outnyttjad kapacitet

Den angivna kapaciteten vid allokeringstillfället kanske är vald för kunna hantera vissa tillfällen då hela kapaciteten måste utnyttjas till fullo. Detta innebär att vid övriga tillfällen kommer det finnas outnyttjad kapacitet. Det som skulle vara intressant att undersöka är huruvida det är möjligt att kunna tillåta viss användning av den outnyttjade kapaciteten på liknande sätt som exempelvis bakgrundsaktiviteter i ett operativsystem som körs då kapacitet finns tillgänglig.

6.2.3 Begränsningsmöjligheter i externa gränssnittet

Som DRAP beskrivs i denna rapport kan användaren ej begränsa utan endast avallokera kanaler. Det skulle vara intressant att utöka det externa gränssnittet med exempelvis en funktion som reallocateChannel(id, set<{capacity,utility}>) som gör det möjligt för applikationen att förändra vilka alternativ en kanal kan använda sig av.

När kan denna funktion vara användbar och hur påverkas schemaläggningen då

funktionen anropas av applikationen under körning?

6.2.4 Dubbelriktade kanaler

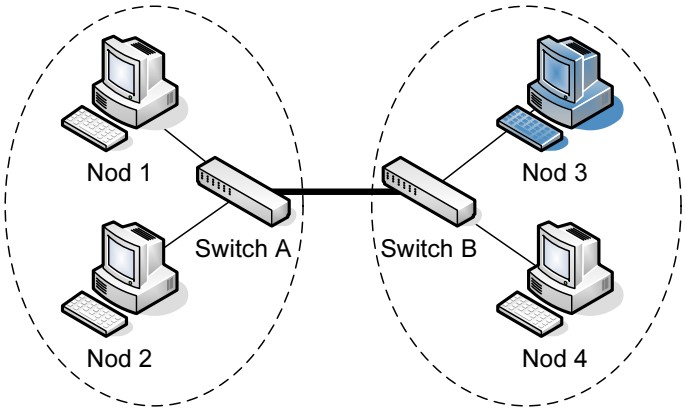
När applikationen allokerar kapacitet skapa endast en kanal åt en riktning, dvs. en enkelriktad kanal. Det kan vara önskvärt att i vissa fall kunna göra en dubbelriktad kanal som kanske kan bidra till minskad trafik på nätverket genom att en av noderna gör en allokering av den kanal åt båda riktningarna med bara ett allokeringsmeddelande istället för att båda gör separata allokeringar av kanaler. Detta bidrar till en optimering i de fall då två noder vill göra allokering av kanaler mellan varandra i båda riktningarna.

6.2.5 Broadcast/Multicast

Skulle vara intressant att se om det är möjligt att göra allokeringar som fungerar som broadcast eller multicast och i vilka sammanhang dessa skulle vara intressanta. Med denna funktionalitet skulle det vara möjligt för applikationen att skicka data till en grupp av noder eller samtliga noder på en och samma gång istället för att skicka samma data till var och en. Det är också viktigt att tänka på vad som bör göras om en nod eller flera noder i en grupp kraschar och hur detta påverkar systemets fortsatta funktion.

6.2.6 Hantering vid användning av flera switchar

Som DRAP fungerar nu är det bara vara möjligt att schemalägga kanaler mellan noder som är anslutna till samma switch. För att kunna ha kontroll på kanaler som allokeras mellan två switchar kräver detta att alla inblandade noder anslutna till dessa switchar kommer överens om ett gemensamt beslut.



Figur 23: Exempel vid användning av flera switchar.

Figur 23 visar ett exempel på hur det kan se ut när två switchar används. Om nod 1 skulle upprätta en kanal till nod 3 kommer nätverkstrafik att flöda från switch A till switch B. Detta innebär att nod 2 måste få veta att nod 1 har upprättat en kanal till nod 3 eftersom annars kan nod 2 upprätta en kanal till nod 3 eller nod 4 och kan därmed överbelasta kommunikationslänken mellan switch A och switch B. När det kommer till noden som ser kanalen som en inkommande kanal räcker det endast att denna nod vet om detta.

Det som är intressant att undersöka är hur mycket som måste förändras i DRAP för att kunna hantera flera switchar eftersom det då är viktigt att ha kontroll på vad för

kanaler som är allokerade mellan switcharna så att inte det kommunikationslänken blir överbelastad. Hur kommer den trafik som DRAP genererar vid allokering och avallokering av kanaler att förändras? Vad för fel bör systemet vara tolerant för etc.?

6.2.7 Förändring av parametrar i det externa gränssnittet

Som det externa gränssnittet ser ut i DRAP i denna rapport, tar allocateChannel som inparameter hur stor datamängd/tidsenhet som ska kunna skickas. En annan variant skulle vara att istället skicka in hur mycket data som ska skickas och dess deadline och då låta DRAP avgöra hur mycket kapacitet som måste allokeras på nätet för att kunna hålla denna deadline.

Om gränssnittet skulle se ut på detta sätt, har det då någon betydelse om deadlines anges i absolut eller relativ tid? På vilket sätt förändras schemaläggningen i DRAP? Vad finns det för för- och nackdelar med att ange deadlines och vilka typer av applikationer skulle ha användning för ett sådant gränssnitt? Hur mycket trafik kommer att genereras? Exempelvis kan det bli många allokeringsmeddelanden om man måste sätta upp kanaler för varje meddelande vilket självklart beror på applikationen. För en FTP-server med stora filer kanske det kan funkar väldigt bra, men om man skulle vara tvungen att göra det för varje transaktion i en realtidsdatabas (RTDB) så blir antagligen overheaden väldigt stor.

6.2.8 Global optimering

I DRAP kan det inte ske något global optimering under allokering av kanal. Med detta menas att när varje nod beslutar om att begränsa eller avallokera kanaler för att optimera dess totala utility kommer detta bara ge en lokal optimering, ingen global optimering av total utility. Detta är på grund av att varje nod inte har någon vetskap om vad andra noder behöver göra för begränsningar eller avallokeringar för att klara att upprätta nya kanaler. Det vore intressant och se om det finns någon sätt som kan lösa detta och få noderna att kunna göra beslut genom att se hur det påverkar hela systemet.

6.2.9 Förbättrad feltolerans

I denna rapport antas det att en kommunikationslänk aldrig genererar fel eller kraschar. Detta antagande är inte realistiskt i verkligheten och bör på något sätt hanteras. Ett möjligt sätt för en nod att upptäcka att det existerar en kraschad kommunikationslänk i systemet kan vara att använda sig av ett heartbeat-protokoll, liknande det som beskrivs i Aguilera m.fl. (1997). När en nod upptäcker att den inkommmande eller utgående kommunikationslänken har kraschat är det i huvudsak viktigt att stoppa den utgående trafiken som kan generera överbelastningar av systemet. Om bara den utgående kommunikationslänken skulle fungera och noden har hårda utgående kanaler är det intressant att undersöka om det är möjligt att bara begränsa dessa kanaler till dess minimala kapacitet istället för att ta bort dem som är något som inte alls är bra. Ett problem som kan uppstå när kommunikationslänken mellan två noder kraschar är att begränsningar eller avallokeringar av kanaler som gjorts kan orsaka att noderna har kvar kanaler som inte längre används efter kraschen av någon av noderna. Hur kan detta problem lösas och göra det möjligt för noder att få reda på vilka kanaler som inte längre används?

För att kunna tolerera hårdvarufel som då switchar kraschar kan redundant hårdvara användas (Bartlett m.fl., 1986). Ett redundant nätverk med en switch kan då användas

som kan ta över all kommunikation mellan noderna. Om en switch slutar att fungera gäller det på något sätt upptäcka att det verkligen är switchen som kraschat och en lösning till detta kan vara att använda sig av det redundanta nätverket. Detta är något som är intressant att undersöka vidare på och om det är möjligt att använda det redundanta nätverket för att upptäcka fel snabbare eller på något annat sätt öka systemets feltolerans mer.

6.2.10 Multipla nätverk

Genom att introducera multipla nätverk vore det också intressant att undersöka om detta kan förbättra realtidskommunikationen i DRAP på liknande sätt som det gjorde i Kao och Garcia-Molina (1992). Ett sätt att fördela trafiken som genereras av noderna är att det ena nätverket endast används för trafik som genereras av DRAP medan det andra nätverket används för resterande trafik. Hur kommer feltoleransen och komplexiteten i DRAP att påverkas vid introduktion av multipla nätverk? Vad finns det mer för sätt att fördela trafiken på mellan nätverken? Kommer kostnaden för hårdvaran fortfarande bli lägre i jämförelse med att använda nätverk som är gjorda för realtidstrafik?

Referenser

Aguilera, M. K., Chen, W. och Toueg, S. (1997), Heartbeat: A timeout-free failure detector for quiescent reliable communication, *in* 'Proceedings 11th International Workshop on Distributed Algorithms', Springer-Verlag, s. 126–140.
URL: *citeseer.ist.psu.edu/article/aguilera97heartbeat.html*

Andler, S., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M. och Eftring, B. (1996), 'Deeds: Towards a distributed and active real-time database systems', *ACM SIGMOD Record* **15**(1), 38–40.
URL: *citeseer.ist.psu.edu/andler96deeds.html*

Andler, S., Hansson, J., Mellin, J., Eriksson, J. och Eftring, B. (1998), 'An overview of the deeds real-time database architecture'. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'98), Orlando, Florida.
URL: *citeseer.ist.psu.edu/355725.html*

Baker, T. (1991), 'Stack-based scheduling of real-time processes', *Real-Time Systems Journal* **3**(1), 67–99.

Bartlett, J., Gray, J. och Horst, B. (1986), Fault tolerance in tandem computer systems, Teknisk rapport TR-86.2, Tandem Computers.
URL: *http://www.hpl.hp.com/techreports/tandem/TR-86.2.pdf*

Blomdell, A., Årzén, K. och Martinsson, A. (2004), Throttlenet: Hard real-time communication using switched ethernet. Opublicerad teknisk rapport, Institutionen för reglerteknik, Lund: Lunds tekniska högskola.

Burns, A. och Wellings, A. (2001), *Real-time Systems and Programming Languages*, Addison-Wesley, Harlow, England.

Chiueh, T. och Venkatramani, C. (1994), Supporting real-time traffic on ethernet, *in* 'Proceedings of IEEE Real-Time Symposium', s. 282–286.
URL: *www.ecsl.cs.sunysb.edu/tr/TR6.ps.Z*

Chiueh, T. och Venkatramani, C. (1997), Fault handling mechanisms in the rether protocol, *in* 'Pacific Rim International Symposium on Fault Tolerant Systems'.
URL: *www.ecsl.cs.sunysb.edu/tr/TR29.ps.Z*

Coulouris, G., Dollimore, J. och Kindberg, T. (1994), *Distributed Systems: Concepts and Design*, 2 edn, Addison-Wesley, Harlow, England.

Coulouris, G., Dollimore, J. och Kindberg, T. (2001), *Distributed Systems: Concepts and Design*, 3 edn, Addison-Wesley, Harlow, England.

Elmasri, R. och Navathe, S. B. (2000), *Fundamentals of Database Systems*, 3 edn, Addison Wesley.

Gray, J. N. (1978), 'Operating systems: an advanced course'. In Operating systems an advanced course, 60, 393-481. Berlin: Springer-Verlag.

Halsall, F. (2001), *Multimedia Communications : Applications, Networks, Protocols and Standards*, Addison-Wesley, Harlow, England.

Hansson, J., Son, S., Stankovic, J. och Andler, S. (1998), Dynamic transaction scheduling and reallocation in overloaded real-time database systems, *in* 'Proceedings of the 5th Conference on Real-Time Computing Systems and Applications (RTCSA'98)', IEEE Computer Press, s. 293–302.
URL: *www.cs.virginia.edu/ stankovic/psfiles/overload.ps*

Hoang, H., Jonsson, M., Hagström, U. och Kallerdahl, A. (2002), Switched real-time ethernet with earliest deadline first scheduling - protocols and traffic handling, *in* 'Proceedings of the 10th Intl. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS 2002)', Fort Lauderdale, Florida, USA.
URL: *www2.hh.se/staff/hoang/papers/WPDRTS01.pdf*

ISO-11898 (1993), 'Road vehicles - interchange of digital information - controller area network (can) for high-speed communication', International Standards Organisation (ISO). Teknisk rapport.

Kao, B. och Garcia-Molina, H. (1992), 'Soft real-time communication over dual non-real-time networks (extended abstract)'.
URL: *http://citeseer.ist.psu.edu/kao92soft.html*

Kopetz, H. (1997), *Real-time systems : Design principles for distributed embedded applications*, Kluwer Academic Publishers, Massachusetts, USA.

Liu, C. L. och Layland, J. W. (1973), 'Scheduling algorithms for multiprogramming in a hard real-time environment', *Journal of the Association for Computing Machinery* **20**(1), 46–61.

Loeser, J. och Härtig, H. (2004), Low-latency hard real-time communication over switched ethernet, *in* 'Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004)', Catania, Italien.
URL: *os.inf.tu-dresden.de/papers_ps/loeser_ecrts2004.pdf*

Martinsson, A. (2002), Scheduling of real-time traffic in a switched ethernet network, Magisterarbete, Lunds tekniska högskola, Institutionen för reglerteknik, Lund. LUTFD2/TFRT5683-SE.
URL: *www.control.lth.se/publications/msc/2002/documents/5683.pdf*

Mathiason, G. (2002), Segmentation in a distributed real-time main-memory database, Magisterarbete, Högskolan i Skövde, Institutionen för datavetenskap, Skövde. HS-IDA-MD-02-008.
URL: *www.ida.his.se/ida/htbin/exjobb/2002/HS-IDA-MD-02-008*

Mathiason, G. och Amirijoo, M. (2004), Real-time communication through a distributed resource reservation approach, Teknisk rapport HS-IKI-TR-04-004, Instutionen för Kommunikation och Information, Högskolan i Skövde.
URL: *www.his.se/upload/19352/technical-report-HS-IKI-TR-04-004.pdf*

Oxford Reference Online (2005). [Tillgänglig på Internet: 2005-02-08].
URL: *www.oxfordreference.com/*

Peterson, L. L. och Davie, B. S. (2000), *Computer Networks : A Systems Approach*, Morgan Kaufman Publishers, San Francisco, USA.

PROFIBUS (2005). [Tillgänglig på Internet: 2005-03-22].
URL: *www.profibus.com/*

Varadarajan, S. och Chiueh, T. (1998), Ethereal: A host-transparent real-time fast ethernet switch, *in* 'Proceeding of International Conference on Network Protocols (ICNP)'.
URL: *www.ecsl.cs.sunysb.edu/tr/TR45.ps.Z*

Venkatramani, C. och Chiueh, T. (1997), Design and implementation of a real-time switch for segmented ethernets, *in* 'International Conference on Network Protocols (ICNP)'.
URL: *www.ecsl.cs.sunysb.edu/tr/TR26.ps.Z*

Wikipedia (2005). (uppdaterad 2004-05-27). [Tillgänglig på Internet: 2005-04-06].
URL: *www.wikipedia.org/*