Forord

Denne rapporten er resultatet av diplomoppgaven som avslutter masterutdanningen ved Norges teknisk-naturvitenskapelige universitet - NTNU. Diplomoppgaven, "Referanseimplementering av Checkpoint Service API ved bruk av Heartbeat" er utført ved NTNU, Fakultet for informasjonsteknologi, matematikk og elektronikk, Institutt for datateknikk og informasjonsvitenskap. Temaet for oppgaven ble foreslått av Øystein Torbjørnsen ved Sun Microsystems i Trondheim, og er en videreføring av fordypningsprosjektet som ble utført forrige semester. Faglærer for prosjektet har vært Svein-Olaf Hvasshovd, som sammen med Øystein Torbjørnsen har vært veileder. Oppgaven ble presentert på *High Performance Computing*-konferansen NOTUR 2005 i Trondheim, hvor den vant prisen for beste poster.

Oppgavetekst:

Per i dag eksisterer det ingen portable løsninger for håndtering av feiltolerante systemer. AIS (Application Interface Specification) definerer et slikt rammeverk, men det foreligger ennå ingen fullstendige implementasjoner av denne standarden. Formålet med denne oppgaven er å lage en referanseimplementering av utvalgte deler av AIS, samt benytte denne til å utvikle en enkel applikasjon som utfører en tjeneste. Applikasjonen skal kjøre på to noder, og checkpointe sine tilstander fortløpende. Ved et nodekrasj vil det da være mulig for andre noder i clusteret å ta over utførelsen av oppgavene til noden som feilet. Dette skal foregå meget raskt, slik at brukeren får opplevelsen av kontinuerlig tilgjengelighet. Clusterprogramvaren Heartbeat skal benyttes i forbindelse med å detektere hvilke noder som feiler.

Vi vil gjerne benytte anledningen til å takke Øystein Torbjørnsen og Svein-Olaf Hvasshovd for god hjelp og nyttige innspill under gjennomføringen av prosjektet. I tillegg vil vi takke Per Kristian Johansen for tilbakemeldinger vedrørende rapporten, og Øyvind Møll for assistanse med Linux. En stor takk går også til Anne Cathrine Elster som gav oss muligheten til å delta på NOTUR 2005.

Trondheim, den 9. juni 2005,

Anja K. Lønningen

Ingunn Lund

Sammendrag

Det eksisterer per i dag ingen implementasjon av standard grensesnitt for å tilby høytilgjengelige systemer i kombinasjon med standard maskinvareplattformer, mellomvare og tjenesteapplikasjoner. Dette medfører at programvaren må tilpasses ulike plattformer og ressurser som er tilgjengelige. Individuell tilpasning resulterer i økte kostnader og tidsforbruk, samt større usikkerhet i forbindelse med utvikling. Service Availibilty Forum presenterer *Application Interface Specification* (AIS) som er et standard grensesnitt for høytilgjengelige løsninger. Rapporten beskriver en prøveimplementasjon av *Checkpoint Service*, som er en del av AIS standarden. Det er også vektlagt hvordan clusterprogramvaren Heartbeat kan benyttes for å oppnå høytilgjengelighet i et distribuert system.

Rapporten beskriver først høytilgjengelige systemers natur, i tillegg til at begreper som feiltoleranse og clusterløsninger introduseres og diskuteres.

Deretter beskrives noe av den høytilgjengelige programvaren som eksisterer på markedet idag. Behovene for en standard diskuteres, og rapporten gir en oversikt over delvise implementasjoner av AIS som foreligger per idag.

Rapporten presenterer så en oversikt over AIS med hovedfokus på *Checkpoint Service*, som er den delen av standarden prosjektet implementerer. Det gis en innføring i entitetene knyttet til *Checkpoint Service*, samt at de viktigste egenskapene ved tjenesten beskrives.

Videre gir rapporten en kort oversikt over de ulike teknologiene som er benyttet i prosjektet, før den utførte implementasjonen presenteres. Det er kun *Checkpoint Service* spesifikke metoder og datastrukturer forbundet med disse som her er beskrevet, samt eventuelle begrensninger ved implementasjonen.

Deretter følger en oversikt over testutførelsen, før resultatene fra de gjennomførte testene fremlegges og diskuteres.

Rapporten konkluderer med at løsningen realiserer basisfunksjonene som inngår i *Checkpoint Service*, og at den fungerer på en tilfredsstillende måte. Det er foretatt målinger av både checkpointtider og takeovertider. Resultatene fra disse målingene er meget gode, og tilsier at applikasjonen imøtekommer kravene til høy tilgjengelighet. Det gjenstår imidlertid noe videre arbeid for å få implementasjonen til å fungere med en tilfeldig applikasjon. Dette inkluderer implementasjon av flere funksjoner samt en viss generalisering av løsningen som foreligger.

Innhold

Figurer

Tabeller

Kapittel 1

Innledning

1.1 Problemstilling

Denne rapporten presenterer en referanseimplementering av *Checkpoint Service*, som er en del av Application Interface Specification (AIS) API'et. AIS er et standardisert sett av funksjoner som er definert av Service Availability Forum (SA Forum).

Når det ikke eksisterer standard grensesnitt for tilgjengelighetshåndtering over flere plattformer, må programvaren tilpasses de ulike ressursene som foreligger. Denne individuelle tilpasningen resulterer i økte kostnader, økt tidsforbruk og større usikkerhet i forbindelse med system- og applikasjonsutvikling. Det kan også medføre svakheter i systemene samt mangel på portabilitet over ulike plattformer [3]. Dersom det ikke foreligger et standard grensesnitt må applikasjoner delvis reimplementeres dersom den underliggende infrastrukturen, som tilbyr høy tilgjengelighet, endres. Dette kan forekomme i forbindelse med at applikasjonen flyttes til et nytt operativsystem eller en ny maskinvareplattform. Det oppstår derfor behov for et grensesnitt som tillater applikasjonsprogramvaren å tilby kontinuerlig tjeneste til brukerne, samtidig som de ovenfornevnte kravene imøtekommes. Dette realiseres ved å benytte redundans, som er et nøkkelkonsept innen høytilgjengelighet. Ved å standardisere grensesnittene unngår man å måtte skreddersy applikasjonene til hver enkelt plattform.

AIS er et forslag til et standard grensesnitt for å tilby høytilgjengelige systemer i kombinasjon med standard maskinvareplattformer, mellomvare og tjenesteapplikasjoner. Motivasjonen er at applikasjonsprogramvare skal kunne utvikles uavhengig av en spesiell maskinvare eller systemplattform. Målet er å utvikle et rammeverk og spesifikasjoner for tjenestetilgjengelighet som reduserer utviklingskostnadene og fremskynder leveransetidspunktet for prosjekter som benytter seg av rammeverket. Dette muliggjøres ved å tillate og sikre portabilitet, samt tillate uavhengige valg av mellomvare, produkter og tjenester. Samtidig skal brukernes krav om tilgjengelighet og pålitelighet imøtekommes.

Den vanligste teknikken for å frembringe høytilgjengelige systemer er å ta i bruk redundans. En implementasjon av *Availability Management Framework* (AMF),

som er en del av AIS, er ansvarlig for å kontrollere denne redundansen. Målet er at applikasjonene skal kunne kommunisere direkte med det standardiserte grensesnittet som tilbys i AMF, uten å måtte forholde seg til administreringen av de redundante komponentene. En prøveimplementasjon av dette rammeverket er beskrevet og gjengitt i [4].

Checkpoint Service brukes sammen med AMF for å tillate applikasjonene å inneha tilstander. Ved å lagre applikasjonens tilstand ved jevne mellomrom, kan man ved et eventuelt krasj gå tilbake til tidspunktet før krasjet oppsto og fortsette utførelsen herfra med de lagrede dataene. Dette medfører at applikasjonen kan kjøre videre upåvirket av feilsituasjonen. Dette vil minimere skadeomfanget, og være med å bidra til høy tilgjengelighet. Tilstandsdataene lagres på flere maskiner, slik at de kan hentes frem selv om en av maskinene går ned. Dersom et slikt tilfelle skulle oppstå, vil en annen maskin overta oppgavene til den som krasjet. Det er ønskelig at denne prosessen gjennomføres så raskt som mulig, noe som gir brukeren opplevelsen av kontinuerlig tilgjengelighet.

Utover redundanskontrollen, som håndteres av AMF, og tilstandshåndteringen, som utføres av *Checkpoint Service*, består AIS av et sett med tjenester som blant annet omfatter låsing, meldingssending og medlemskapsinformasjon. Disse er nærmere beskrevet i kapittel 4.3. Implementering av disse tjenestene er imidlertid utenfor rammen av dette prosjektet.

1.2 Bakgrunn for problemstillingen

Oppgaven ble gitt av Sun Microsystems, som er en av aktørene i SA Forum. Ressurser fra firmaet har vært involvert i utviklingen av AIS standarden. Interessen for en implementasjon av denne baserer seg på behov blant flere av firmaets produkter.

Ved dags dato finnes ingen fullstendige AIS implementasjoner som er offentlig tilgjengelige. Det eksisterer imidlertid en rekke systemer som krever høy tilgjengelighet og pålitelighet, blant annet innen database- og telekommunikasjonsverdenen. Resultatet har så langt blitt at hver leverandør av slike feiltolerante systemer har laget sin egen løsning på problemet. Ved hjelp av AIS får man innkapslet tilgjengelighetsproblematikken i en pakke som kan brukes igjen og igjen. Dette vil redusere kostnaden ved å utvikle et system betraktelig.

Det vil i tillegg være interessant å integrere AMF og *Checkpoint Service*, som implementeres i dette prosjektet, med eksisterende feiltolerante systemer. Det er imidlertid lite trolig at leverandører raskt og drastisk vil endre sine produkter slik at de vil fungere med disse tjenestene. Derfor vil en slik integrering kreve at visse mål etterstrebes:

- Egenskapene når det gjelder pålitelighet og tilgjengelighet i en slik integrert løsning bør være på nivå med tilsvarende egenskaper i den opprinnelige løsningen.
- Store forandringer bør ikke være nødvendig, verken i implementasjonen av AMF og *Checkpoint Service*, eller i det eksisterende systemet i forbindelse med integreringen.

- AMF skal være ansvarlig for implementasjon av redundanskontroll.
- *Checkpoint Service* skal være ansvarlig for tilstandshåndtering.

En implementasjon av AIS skal ideelt sett være uavhengig av både underliggende maskinvare og operativsystem, og skal kunne benyttes av alle typer applikasjoner.

1.3 Tidligere arbeider

I 2003/2004 ble det skrevet prosjekt- og diplomoppgave ved NTNU av Einar Hartvik Olsen som omhandlet design av AMF. Her lå hovedfokuset på hvilke clusterprogramvarer som var tilgjengelige. Det ble gitt en vurdering av de ulike clusterprogramvarene, samt en implementering basert på disse vurderingene.

I tillegg finnes det *open source* kildekode hvor deler av AIS-standarden er implementert. Disse er nærmere beskrevet i kapittel 3.2.1 og 3.2.2. Det har ikke vært mulig å oppdrive dokumentasjon på en fullstendig implementasjon av AIS standarden.

Høsten 2004 gjennomførte vi en prosjektoppgave hvor det ble utviklet en referanseimplementering av AMF [4]. Implementeringene som ble foretatt er videreført i dette prosjektet. Det er kun foretatt små endringer i den eksisterende koden for å tilpasse løsningen til den gjeldende problemstillingen. Utover dette tar ikke rapporten utgangspunkt i tidligere arbeider. Det er valgt å se bort fra resultater og kode implementert av andre, og heller utviklet en selvstendig implementasjon. På denne måten sikres det at programmet utfører det problemstillingen uttrykker i henhold til kravene vi setter. I tillegg oppnås en dypere forståelse av emnet, samt fullstendig kjennskap til og kunnskap om alle detaljer i programkoden.

1.4 Rapportstruktur

Kapittel 2 tar for seg prinsippene for høytilgjengelighet og feiltoleranse, og gir eksempler som illustrerer disse konseptene. I kapittel 3 beskrives dagens situasjon kort; behovet for en standard og eksisterende implementasjoner av AIS diskuteres, i tillegg til at det presenteres en oversikt over de største aktørene innen høytilgjengelig programvare. Kapittel 4 gir en detaljert systembeskrivelse av AIS og dens funksjonalitet, med vekt på *Checkpoint Service* og dens tjenester. I kapittel 5 beskrives hovedteknologiene som er benyttet i dette prosjektet, deriblant Heartbeat, TCP/IP og UDP. Kapittel 6 forklarer hvordan prosjektets implementasjon er utført, inkludert en detaljert beskrivelse av de viktigste metodene og datastrukturene. I kapittel 7 beskrives utførelsen av testene, mens resultatet fra disse diskuteres i kapittel 8. I kapittel 9 gis en konklusjon, samt forslag til videre arbeid.

1.5 Notasjon

Engelske termer benyttes der det er naturlig. Dette gjelder ord som ofte brukes i norsk fagterminiologi, som standby, cluster og checkpoint. For elementer som er direkte spesifisert i standarden, og som også er brukt i implementasjonen av dette prosjektet, beholdes det opprinnelige navnet for å unngå forvirring rundt bruk av flere begrep for samme element. De aktuelle begrepene er derfor ikke oversatt til norsk. Dette omfatter for eksempel funksjonsnavn, tjenestene AIS standarden tilbyr og attributter hos de ulike komponentene. Som eksempler nevnes funksjonsnavnet *saCkptInitialize()*, tjenesten *Checkpoint Service* og attributtet *retentionDuration*. Alle disse er, i likhet med funksjoner fra andre C-bibliotek, skrevet i kursiv.

Kapittel 2

Høytilgjengelige systemer

Dette kapitlet gir en oversikt over prinsippene for høytilgjengelige systemer, samt hvilket behov det er for slike systemer i dag. I den forbindelse introduseres også terminologi og bakgrunnsteori for høytilgjengelighet og feiltoleranse, samt at det gis en generell oversikt over clusterløsninger og deteksjon av feilede noder.

2.1 Behovet for høytilgjengelige systemer

I dagens samfunn er det en økende etterspørsel etter systemer med minimal nedetid. Internett har, med sin globale natur, bidratt til at bedrifter som banker og nettbutikker ønsker å levere høytilgjengelige tjenester til et økende antall kunder. I en slik kontekst kan nedetid medføre tapt inntekt og misfornøyde brukere. Innen børs og telecom kan man også tenke seg scenarier der systemets nedetid raskt får store økonomiske konsekvenser. Sykehusdrift er en annen næring hvor man er avhengig av kontinuerlig tilgjengelige systemer. Her kan det få dramatiske konsekvenser dersom et system går ned og blir utilgjengelig. Dette gjelder også systemer for kontroll og dirigering av tog og fly, hvor det kan gå tapt mange menneskeliv ved et systemkrasj dersom det ikke eksisterer et backup-system som raskt kan ta over tjenestene. Kritiske systemer som dette må tilby et høyt nivå av tilgjengelighet og pålitelighet.

Et høytilgjengelig system er et system som er designet for å eliminere eller minimere tap av tjenester som følge av planlagte eller ikke-planlagte feil. Høytilgjengelige systemer kan altså, i motsetning til kontinuerlig tilgjengelige systemer, tillate seg å gjøre feil, men de er designet for å være raskt tilbake i tjeneste. Hvor lang tid dette tar vil avhenge av antall servere og datamengden som inngår.

2.2 Definisjon på tilgjengelighet

Pålitelighet er vanligvis definert som "Mean time between failures" (MTBF). Dette er et mål, i antall timer, på hvor pålitelig en komponent er. *Virkelig nedetid*, "Mean time to repair" (MTTR), er et mål på tiden som kreves for å bringe

et feilet system tilbake i tjeneste. *Tilgjengelighet* defineres som sannsynligheten for at et system er oppe til et gitt tidspunkt. Denne størrelsen kan uttrykkes ved hjelp av MTBF og MTTR etter følgende formel:

$$Tilgjengelighet = \frac{MTBF}{MTTR+MTBF}$$

Tilgjengelighet kan altså maksimeres ved å øke tiden mellom feilene og minimere tiden som kreves for å reparere et feilet system.

2.3 Feiltoleranse

Høy tilgjengelighet krever systemer som er designet for å tolerere feil. Det innebærer at systemet må detektere, rapportere og maskere feil, for så å fortsette og tilby sine tjenester mens feilet komponent repareres offline. Systemet må være istand til å tolerere en rekke typer feil, som alle vil stille ulike krav til detektsjon, rapportering, maskering og reparering. Feil som oppstår kan skyldes defekt maskinvare, feil i programvaren, operatør- eller vedlikeholdsfeil eller påvirkning fra ytre omgivelser. Tabell 2.1 viser noen eksempler på feil som kan oppstå.

Maskinvarefeil	• avmagnetisering av disk • brudd i nettverk • defekt minne
Programvarefeil	• minnet blir fullt • komme utenfor gyldig verdi av en løkkevariabel • en ufullstendig melding mottas
Omgivelsesfeil	• strømbrudd • brann • oversvømmelse
Vedlikeholdsfeil	• feil ved installasjon av ny programvare eller maskinvare • blande programvareversjoner som ikke er kompatible
Operatørfeil	• la en fil bli så stor at den ikke får plass på disken • glemme å lagre slik at man får inkonsistente data

Tabell 2.1: Eksempler på feil som kan forårsake nedetid hos systemer

Avhengig av hvilken type feil man ønsker å tolerere, tas det i bruk ulike teknikker.

Gray og Siewiorek [1] introduserer følgende 5 nøkkelkonsepter for å bygge høytilgjengelige systemer:

- modularitet - systemet dekomponeres i moduler som hver for seg:
  - tilbyr en veldefinert tjeneste
  - er designet slik at feil i modulen ikke forårsaker feil i andre moduler
  - kan erstattes av en ny modul dersom den feiler
- *fail-fast* moduler - dersom modulens oppførsel avviker fra den korrekte oppførselen skal modulen stoppes
- uavhengige feilmodi - moduler og forbindelser mellom disse designes slik at feil i en modul ikke affekterer andre moduler
- redundans - duplikater av moduler sørger for at en modul kan erstattes øyeblikkelig når feil oppstår
- reparering - når en modul har feilet er det helt avgjørende for videre tjenesteytelse at den feilende enhet repareres offline for så å reintegreres i systemet, slik at man til enhver tid har minst en reservemodul

Prinsippene nevnt ovenfor kan anvendes på design av maskinvare såvel som programvare, og vil til sammen kunne bidra til å maskere de fleste typer feil.

Sett i et historisk perspektiv bidrar maksinvare i dag til en langt mindre andel av alle feil som forekommer sammenlignet med tidligere. Dette skyldes bedre design og ny teknologi, for eksempel fiberoptikk, som har gjort maskinvaren mer pålitelig. Maskinvarens nye design har dessuten ført til at behovet for vedlikehold er blitt kraftig redusert. I dag er det derfor operatør- og programvarefeil som er den dominerende bidragsyteren til systemenes nedetid. Når det gjelder feil forårsaket av programvare, skiller man mellom såkalte persistente og transiente feil. Persistente feil er feil som vil opptre hver gang programmet kjøres, mens transiente feil er feil som vil forsvinne dersom operasjonen kjøres igjen i en annen kontekst. Videre vil omgivelsesfeil alltid innebære en utfordring; selv om hyppigheten av feil forårsaket av for eksempel strømbrudd eller brann er lav, er muligheten tilstede og utgjør en faktor man må ta høyde for. Det viktigste prinsippet for å maskere alle de nevnte typene av feil er redundans. Både maskinvare og programvare kan repliseres og på denne måten danne grunnlaget for såkalte prosesspar (se kapittel 2.3.2) og systempar (se kapittel 2.3.3).

2.3.1 Feilklasser

I forbindelse med feiltolerante distribuerte systemer introduseres begrepene feilklasser og feilklassifisering [2]. Hver server har en tjenestespesifikasjon som sier noe om hvordan den skal oppføre seg, både i forhold til hva slags respons den skal gi til brukeren og tidsintervallet den skal svare innen. En server som er designet for å tilby en spesiell tjeneste har *riktig* oppførsel hvis den, i henhold til input, oppfører seg på en konsistent måte i forhold til tjenestespesifikasjonen. En *feil* oppstår dersom serveren ikke oppfører seg som spesifisert. Det finnes ulike typer feil som klassifiseres på følgende måte:

- Utelatelsesfeil

  Feil som oppstår når serveren ikke gir respons på forespørsler. Eksempler på dette kan være en kommunikasjonstjeneste som mister enkelte meldinger, eller en transaksjonsserver som aborterer transaksjoner.

- Timingfeil

  Serveren svarer korrekt på forespørselen, men utenfor det spesifiserte tidsintervallet. Siden svaret kommer for sent eller for tidlig, kan dette også kalles en ytelsesfeil. Eksempler på dette er en prosessor som starter for tidlig grunnet for rask timer, eller en transaksjonsserver som ikke har *real-time* responstid på grunn av overlast.

- Responsfeil

  Serveren gir galt svar eller gal tilstandsendring. Databasen vil da være inkonsistent i forhold til svaret man får. Eksempler på responsfeil er at DBMS finner et slettet tuppel eller ikke finner et innsatt tuppel.

- Krasj

  Dersom serveren ikke svarer på noen henvendelser før den restartes, har det oppstått et krasj. Det er fire underklasser av denne feilen (se figur 2.1):

  - Amnesia krasj

    Serveren restartes i en predefinert, initiell tilstand. Tilstanden avhenger ikke av input gjort før krasjet fant sted. Et eksempel på dette er at operativsystemet restartes som følge av en *reboot*.

  - Delvis amnesia krasj

    Ved restart er noen deler av tilstanden den samme som før krasjet, mens resten er satt til en predefinert, initiell tilstand. Dette kan for eksempel være et filsystem hvor bufferet krasjer. Alt som lå i bufferet vil da gå tapt, mens dataene som var lagret på disk fremdeles kan aksesseres.

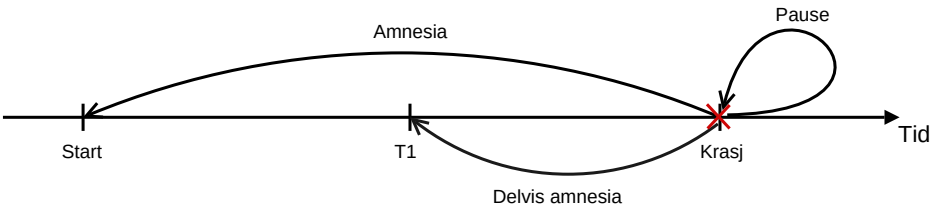
  - Pausekrasj

    Serveren vil i dette tilfellet bli restartet i tilstanden den hadde rett før krasjet. Dette kan oppstå ved et DBMS systemkrasj etterfulgt av en recovery av DBMS tilstanden som reflekterer alle *commit*ede transaksjoner ved krasjtidspunktet.

  - Stopp-krasj

    Ved denne typen krasj vil serveren aldri starte opp igjen.

Ved programmering av hendelsesforløpet til *recovery* etter en serverfeil, er det viktig å vite hvilken feiloppførsel serveren viser. Hvilke hendelser som skal inntreffe avhenger av serverens feiloppførsel. I et feiltolerant system utvider man derfor serverspesifikasjonen til å inneholde feilsemantikk. Dersom spesifikasjonen av server r tilsier at den observerte feiloppførselen er innenfor klasse F, heter det at "r har F feilsemantikk". Det innføres også en klasse S som beskriver serverens standardoppførsel, eller feilfrie semantikk. Målet er å få sannsynligheten for at server r får en feil utenfor en gitt feilklasse F så liten at denne blir neglisjerbar.



Figur 2.1: Underklasser av krasj

I den forbindelse er det avgjørende om man kan anta at de eneste feilene r får er inneholdt i feilklasse F.

Spesifikasjonen av server r må bestå av funksjonelle krav S_r og F_r (serverens standard semtantikk og feilsemantikk), i tillegg til en stokastisk spesifikasjon. De stokastiske kravene bør gi en minste sannsynlighet s_r for at standardoppførselen S_r observeres under kjøring. En tjeneste er tilgjengelig når minste standard oppførselskrav er oppfylt. Kravene bør også spesifisere en største sannsynlighet c_r for at en (potensielt katastrofal) feil, som ikke er beskrevet i feiloppførselen F_r, observeres. Ut i fra dette kan man beregne sannsynligheten for at man er utenfor semantikken beskrevet for server r, det vil si sannsynligheten for at oppførselen til server r er utenfor $S_r \cup F_r$. Dette kan skje dersom det oppstår feil man ikke har tatt høyde for i spesifikasjonene. Responsfeil og tilstandsfeil vil normalt være garantert for i feilsemantikken. Dersom disse feilsituasjonene allikevel oppstår, for eksempel grunnet en programmeringsfeil, er man også utenfor $S_r \cup F_r$. Dette er noe man aldri kan garantere seg mot, men sannsynligheten for at en slik situasjon skal oppstå er veldig liten (typisk 10^{-9}).

2.3.2 Prosesspar

Et prosesspar består av to prosesser som kjører samme programvare på hver sin maskinvare. Prosessene kobles til klienter, til hverandre og til lagringsmedium i doble nettverk slik at man unngår *single-points-of-failure*. De to prosessene må kjøre med separat CPU og minne, ha uavhengig energiforsyning og i tillegg være *fail-fast*. Den ene prosessen tildeles rollen som primær og denne vil i utgangspunktet være ansvarlig for å levere den aktuelle tjenesten. Den andre prosessen blir en standbyprosess som opptrer som en slags reserve for primærprosessen. Dersom primærprosessen av en eller annen grunn feiler, står standbyprosessen klar til å ta over fra det punktet hvor primærprosessen sluttet. Dette kalles *takeover* og involverer to viktige aspekter som må håndteres:

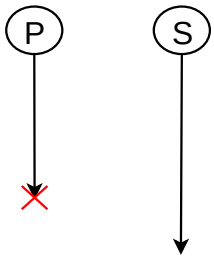
- detektering - standby må oppdage at primær har feilet
- fortsettelse - før standby overtar kjøringen må den vite tilstanden primær var i da den feilet

Prosesspar kan maskere maskinvare-, programvare- og vedlikeholdsfeil. Prosesspar kan også maskere enkelte typer miljøfeil, som for eksempel strømbrudd, men vil ikke ha noen effekt på operatørfeil.

Prosesspar kan realiseres på flere måter, og de ulike designmetodene er beskrevet nedenfor:

1. Lockstep

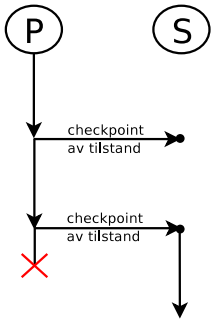
   Primær og standby utfører samme sett av instruksjoner synkront på to uavhengige prosessorer. Dette designet gir god toleranse for feil på maskinvare, men tolererer hverken persistente eller transiente programvarefeil.



Figur 2.2: Lockstep

2. State checkpointing

   Primær utfører instruksjonene og sender tilstandsendringer til standby. Ved *takeover* begynner standby å kjøre fra siste checkpoint. Dette gir meget god feiltolereanse, men er krevende å programmere ettersom det kan være vanskelig å få med alle de variablene som bør sendes over i checkpointet.



Figur 2.3: State checkpointing

3. Automatisk checkpointing

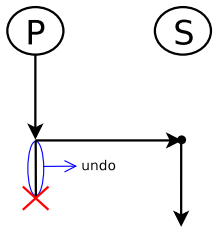
   Kompilator og operativsystem analyserer programmet, identifiserer hva som representerer tilstanden og bestemmer når checkpointing skal skje. Tilstanden sendes over hver gang den endres. Ulempen med denne teknikken er at man får stort overføringsvolum av data ettersom tilstanden i sin helhet overføres i hvert checkpoint i stedet for at kun tilstandsendringene sendes.

4. Delta checkpointing

   Dette er en videreutvikling av State checkpointing, men her er det de logiske i stedet for fysiske oppdateringene som overføres i checkpointet. Dette gir bedre ytelse enn Automatisk checkpointing siden mindre data overføres, men metoden er, på samme måte som State checkpointing, vanskelig å implementere.

5. Persistente prosesspar

   Her er prosessene tilstandsløse. Når primær går ned, våkner standby opp i en nulltilstand med amnesia om hva som skjedde idet primær feilet. Bare åpning og lukking av sesjoner blir checkpointet til standby under kjøring. Persistente prosesser er svært enkle å programmere, men maskerer i seg selv ikke feil. Derfor bruker man ofte persistente prosesser sammen med transaksjoner for å synkronisere tilstanden til primær og standby. Denne kombinasjonen maskerer både feil på maskinvare og transiente feil i programvaren.



Figur 2.4: Persistente prosesspar

2.3.3 Systempar

Et systempar består av to identiske systemer som er plassert med stor geografisk avstand. Systemene er på ulike kommunikasjonsnett og strømnett og har uavhengige operatører. Systempar kan dermed maskere både miljø-, vedlikeholds- og operatørfeil. Systemets klienter er *in-session* med begge systemene, men sender jobben til kun ett system. Dersom det ene systemet feiler, overtar det andre. Transaksjonsmekanismen benyttes for å rydde opp i tilstander ved *takeover*. For såkalte *one-safe* systemer kan en transaksjon *commit*e på primærsystemet før loggen sendes over til standby og de samme operasjoner utføres her. For *two-safe* systemer kreves det imidlertid at standby har mottatt loggen og lagret denne på disk før både primær og standby *commit*er.

2.4 Høytilgjengelige clusterløsninger

Et høytilgjengelig cluster realiserer ideen med prosesspar. Clusteret består av en gruppe av datamaskiner (noder) som samarbeider for å tilby tjenester. Tjenestene tilhører ikke en enkelt node, men clusteret som helhet. Dette fungerer slik at dersom en enkelt node i clusteret feiler, forårsaker ikke dette at tjenesten blir

utilgjengelig. Nodens oppgaver vil da raskt og automatisk utføres av en annen node innad i clusteret. Dersom en slik løsning implementeres i et system vil nedetiden reduseres betraktelig, og dermed bidra til økt tilgjengelighet (se kapittel 2.2). Høytilgjengelige clustre minimerer avbrytelser i systemet ved å raskt bytte over fra et feilet system til et kjørende system. Dette gir brukeren illusjonen av kontinuerlig tilgjengelighet.

Et av elementene i høytilgjengelig ytelse er tiden en konfigurasjon bruker på å komme tilbake i systemet etter en feil, det vil si hvor lenge systemet kan være utilgjengelig før det merkes av brukerne. For å bestemme den totale tiden dette begrenses av, må man inkludere tid til feildeteksjon, notifisering og recovery. Feildeteksjon er typisk implementert ved at nodene utveksler I'm alive-meldinger med hverandre, også kalt heartbeats (hjerteslag). Det er nødvendig å detektere når en node går ned, slik at de andre nodene i clusteret kan overta dens oppgaver. Dette løses ved å bruke en clusterprogramvare som detekterer om nodene er i live, og som formidler denne statusen til alle nodene i clusteret. Programmet sender typisk ut pakker fra hver node i clusteret til alle de andre for å indikere at noden er operativ. Dersom det ikke mottas pakker fra en node innen et gitt tidsintervall, antar man at noden har sluttet å utføre tjenester grunnet en feilsituasjon. Når en slik feil detekteres, utføres en *recovery* prosedyre kalt *failover*. Den feilede noden deaktiveres og en substituttnode tar over dens arbeidslast. Tiden som går med til *failover*, det vil si tiden fra en node feiler til en annen har tatt over dens tjenester, avhenger av antall steg og kompleksiteten av stegene som kreves for å initialisere spesifikke applikasjoner som kjørte på den feilede noden. Clusterprogramvaren benyttet i dette prosjektet er nærmere beskrevet i kapittel 5.1.

Kapittel 3

State of the Art

Dette kapitlet forklarer hvorfor det er behov for et standardisert grensesnitt som AIS. Videre beskrives ulike eksisterende implementasjoner av AIS, og det gis et innblikk i hvor langt de ulike aktørene har kommet i implementeringsfasen. Deretter følger en oversikt over høytilgjengelig programvare som benyttes i ulike clusterløsninger per i dag.

3.1 Behovet for en standard

Behovet for et standard grensesnitt drives av kravene fra nettverkstjenester som er i rask utvikling. Tradisjonelt sett har data- og kommunikasjonstjenester blitt utviklet på rettighetsbeskyttede plattformer som måtte tilfredsstille meget spesifikke krav til tilgjengelighet, pålitelighet, ytelse og responstid [7]. Tilbyderne av kommunikasjonstjenester blir nå utfordret til å kostnadseffektivt møte behovene for nye arkitekturer, tjenester og økt båndbredde. Dette skal skje parallelt med at systemene forblir høytilgjengelige, skalerbare, sikre og pålitelige, samt at de skal være enkle å vedlikeholde og oppgradere. Brukere forventer at nye tjenester skal leveres på forespørsel og uten avbrytelser i det eksisterende systemet. Dette betyr at tjenestene må ha høyest mulig nivå av tilgjengelighet og pålitelighet, samtidig som dette balanseres opp mot kravene om en kort utviklingssyklus og lave utviklingskostnader.

Spørsmålet blir hvordan man kan møte disse kravene ved bruk av eksisterende infrastrukturer og teknologier. Rettighetsbeskyttede plattformer vil bli for dyre å utvikle, og disse mangler ofte støtte for eksisterende og kommende standarder. Trenden går mot å levere neste generasjons kommunikasjonstjeneste ved bruk av åpne standardiserte, høytilgjengelige plattformer. Et uniformt, åpent programvare miljø kombinert med kommersielle hyllevareprodukter innen programvare og maskinvare, er en nødvendig del av disse nye arkitekturene. Åpne plattformer forventes å redusere kostnadene og risikoen ved utvikling og levering av blant annet telekommunikasjons tjenester, tillate raskere *time to market* og sikre portabilitet og samspillsevne. En effektiv løsning på dette vil være en bred tilpasning av åpne standarder. SA Forum har utviklet standarder for bruk i høytilgjengelige applikasjoner. Disse er nærmere beskrevet i kapittel 4.1 og 4.2.

Når det ikke eksisterer et standardisert grensesnitt for å håndtere tilgjengelighet på tvers av ulike plattformer, må programvaren tilpasses etter ressursene som er tilgjengelige. En slik individuell tilpasning resulterer i økte utgifter og økt tidsforbruk i forbindelse med utvikling av programvare [6]. Det kan også medføre systemsvakhet da produktene stadig må gjenutvikles, og mangel på portabilitet da de spesialtilpasses hvert enkelt tilfelle. Dersom den underliggende infrastrukturen endres, kan man risikere å måtte utføre omfattende endringer på en allerede ferdigutviklet applikasjon. Dette kan for eksempel skje hvis applikasjonen overføres til et nytt operativsystem eller en ny maskinvareplattform. Av grunnene nevnt ovenfor oppstår det derfor behov for et grensesnitt som tillater applikasjonsprogramvaren å tilby kontinuerlig tjeneste til brukerne ved å utnytte redundansen som allerede er tilstede i høytilgjengelige systemer. For å unngå at disse må spesialtilpasses hver enkelt plattform, må grensesnittene standardiseres. Applikasjonsprogramvare kan da utvikles uavhengig av en spesiell maskinvare eller systemplattform. Utviklingskostnader kan dermed reduseres, portabilitet sikres og brukerne kan fritt velge mellom ulike mellomvarer og produkter.

3.2 Eksisterende implementasjoner av AIS

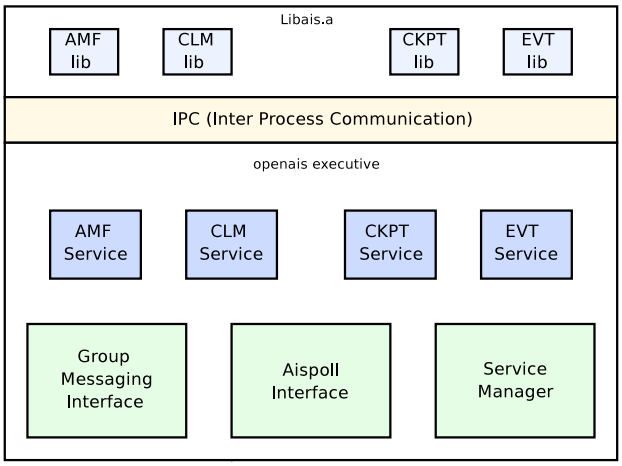
Det er flere aktører som delvis har implementert, eller gjort forsøk på å implementere, SA Forums AIS. Dette inkluderer både *open source* implementasjoner og diplomoppgaver ved NTNU. Dette delkapitlet nevner de største aktørene, samt de med størst betydning for gjennomførelsen av dette prosjektet.

3.2.1 OpenAIS

OpenAIS er en *open source* implementasjon av SA Forums AIS for høytilgjengelige clustre [16]. Denne programvaren gjør det mulig å utvikle redundante applikasjoner som tolererer feil i maskinvare, operativsystem og applikasjoner. OpenAIS skal tilby 100% feilfrie operasjoner med meget gode ytelseskarakteristika. Prosjektet har per idag implementert fire av de seks APIene i AIS-standarden. Disse er *Availability Management Framework* (AMF), *Cluster Membership Service* (CLM), *Checkpoint Service* (CKPT) og *Event Service* (EVT). De delene av standarden som gjenstår før prosjektet kan ferdigstilles er *Message Service* (MSG) og *Lock Service* (LCK). For mer informasjon om de ulike APIene henvises det til kapittel 4.3.

OpenAIS prosjektet ble startet for å utvikle en fullstendig høytilgjengelig clusterløsning basert på grensesnittet til SA Forum. Løsningen bygger på en klient-server arkitektur som er illustrert i figur 3.1. Bibliotekene i OpenAIS implementerer APIene fra SA Forums standard. Deretter linkes bibliotekene inn i ende-applikasjonen, hvorpå de kan etterspørre tjenester fra hovedprosessen (openais executive). Denne bruker så *group message protocol* til å opprette cluster kommunikasjon mellom flere noder. Når en gruppe av noder har tatt en avgjørelse om hvilke oppgaver som skal utføres, sendes et svar tilbake til biblioteket som igjen svarer til bruker-APIet. Alle applikasjoner som skal benytte seg av OpenAIS må linke seg mot Libais.a biblioteket, som fremstilt i figur 3.1. Dette biblioteket

bruker IPC (Inter Process Communication) til å kommunisere med hovedprosessen. Biblioteket er kun ansvarlig for å pakke inn en forespørsel som en melding, og i hver melding er det spesifisert hvilken tjeneste den tilhører. Meldingen sendes tilbake til hovedprosessen som prosesserer den. Biblioteket venter så til det får svar tilbake fra denne.



Figur 3.1: Arkitekturen til OpenAIS [16]

Når biblioteket sender ut en melding via IPC, havner denne hos tjenesten som er spesifisert i meldingen. Deretter videresendes meldingen til alle andre noder i clusteret via multicastfunksjoner i *Group Messaging Interface*.

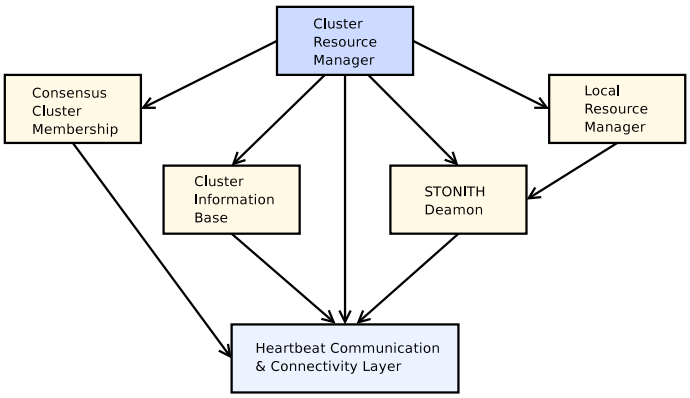
Prosjektet benytter seg av virtuell synkronisering. Dette er en modell for sending av gruppemeldinger, og innehar en rekke fordeler som integrert medlemsskap, pålitelig kommunikasjon, bruk av standard UDP/IP til nettverks multicast, man unngår tofase commit protokoller og ytelsen blir meget bra.

3.2.2 Linux-HA

Linux-HA er det eldste eksisterende høytilgjengelighets prosjektet for Linux. Det har som mål å tilby en høytilgjengelig clusterløsning for Linux som fremmer pålitelighet, tilgjengelighet og tjenester gjennom en felles utviklingsinnsats [17]. Programvaren er mye brukt i ulike prosjekter, og er en viktig komponent i flere høytilgjengelige løsninger. Prosjektet har vært i produksjon siden 1999, og er på det nåværende tidspunkt i bruk på over 10 000 *sites*.

Det har vært jobbet med å innlemme SA Forums AIS i grensesnittet til Linux-HA. Det er ikke implementert like store deler av AIS-standarden som i OpenAIS-prosjektet. I den nåværende versjonen er to av de seks APIene ferdig implementert, selv om de enda ikke fungerer optimalt ytelsesmessig. Dette gjelder *Cluster Membership Service* og *Checkpoint Service* (se kapittel 4.3.1 og 4.3.2). De resterende APIene oppgis å være i en tidlig implementeringsfase. Versjon 2 ble utgitt som en prøve-release i mars 2005. Her skal fem av APIene være inkludert, i tillegg til at applikasjonen er kraftigere enn forrige versjon. Linux-HA er et portabelt

høytilgjengelig clusterprodukt, og portabilitet var et av de viktigste designmålene. Programvaren kan kjøres på FreeBSD og Solaris, i tillegg til Linux.



Figur 3.2: Arkitekturen til Linux-HA Release 2

Heartbeat er en viktig del av Linux-HA prosjektet, og er også benyttet i dette prosjektet. Heartbeat er nærmere beskrevet i kapittel 5.1.

3.2.3 Diplomer og prosjekter ved NTNU

Det ble i 2003/2004 skrevet prosjekt- og diplomoppgave av Einar Hartvig Olsen som omfattet design av AMF. Her ble ulike cluster programvarer diskutert, og en implementering ble foretatt basert på dette grunnlaget. Denne oppgaven bygger ikke videre på denne implementeringen, da arbeidsmetodikken og infallsvinkelen var annerledes enn den som er valgt for dette prosjektet.

Høsten 2004 gjennomførte forfatterne av denne diplomoppgaven en prosjektoppgave som omhandlet samme tema [4]. Det ble her foretatt implementering av deler av AMF, samt utviklet en tilstandsløs applikasjon for å teste programvaren. Implementeringene som her ble foretatt er videreført i dette prosjektet. Det er kun foretatt små endringer i den eksisterende koden for å tilpasse løsningen til å fungere på to noder. I kapittel 6 gis det en inngående beskrivelse av implementasjonene utført i dette prosjektet.

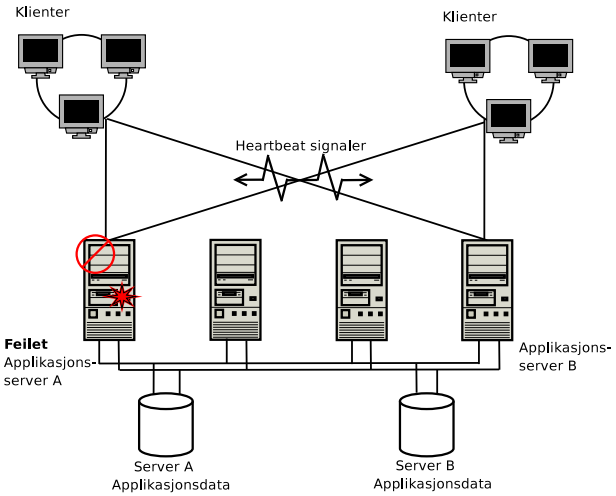
3.3 Eksisterende høytilgjengelig programvare

Det eksisterer mange ulike varianter av programvare som skal sikre høytilgjengelighet for applikasjoner. I dette avsnittet nevnes noen av de største aktørene innen høytilgjengelig cluster programvare som eksisterer på markedet i dag. Systemer for gruppekommunikasjon er designet for å støtte kommunikasjon mellom prosesser som samarbeider i grupper. Systemene tilbyr et underliggende lag som håndterer vedlikehold av medlemsskapet i gruppen og pålitelig meldingssending. Eksempler på slike systemer er Spread, Transis og Horus, og disse er beskrevet under.

3.3.1 LifeKeeper

SteelEyes LifeKeeper for Linux [8] er en programvareapplikasjon som sikrer kontinuerlig tilgjengelighet for applikasjoner. LifeKeeper opprettholder høy tilgjengelighet av et clustret Linux system ved å overvåke systemet og dets "helse". For å tillate automatisk system- og applikasjons *recovery* ved et eventuelt systemkrasj, utfører LifeKeeper *failover* til andre servere i clusteret. Dette minimerer riskoen for enkeltfeil, og lar systemet møte de strenge tilgjengelighetskravene som er nødvendig for å kunne ha et feiltolerant miljø.

LifeKeeper gir Linux-miljøer mulighet for å vedlikeholde feiltoleranse ved å la fungerende servere i clusteret ta over for feilede servere eller applikasjoner. Feil som oppstår i maskinvare eller applikasjoner oppdages via ulike feildetekteringsmekanismer før et eventuelt fullstendig systemkrasj opptrer. Clusteret overvåkes ved bruk av blant annet LAN *heartbeats*; signaler som sendes ut fra en server for å indikere at den fungerer og er i live. Ved at redundante signaler sendes mellom servernodene kan man oppdage nodene som feiler, og dermed går ned. LifeKeeper begrenser også unødvendig *failover*. Dette utføres ved å gjøre *recovery* på feilede applikasjoner uten å gjøre full *failover* til en annen server dersom maskinvaren fremdeles er aktiv.



Figur 3.3: LifeKeeper overvåker systemet ved bruk av heartbeats [9]

3.3.2 Beowulf

Det første Beowulf clusteret ble konstruert i 1994, og ble utviklet spesielt for å tillate høytilgjengelige beregninger ved bruk av kommersielle maskinvare komponenter som kjører et open source operativsystem som Linux eller FreeBSD. På denne måten ble de mye billigere å utvikle enn tradisjonelle superdatamaskiner. Et Beowulf cluster er et skalerbart cluster som fokuserer på å utføre parallelle utregninger [10]. Nodene henger sammen i et raskt nettverk, og er dedikerte til å kjøre høyytelsesoppgaver. Tunge beregninger innen felt som genetikk, fluid

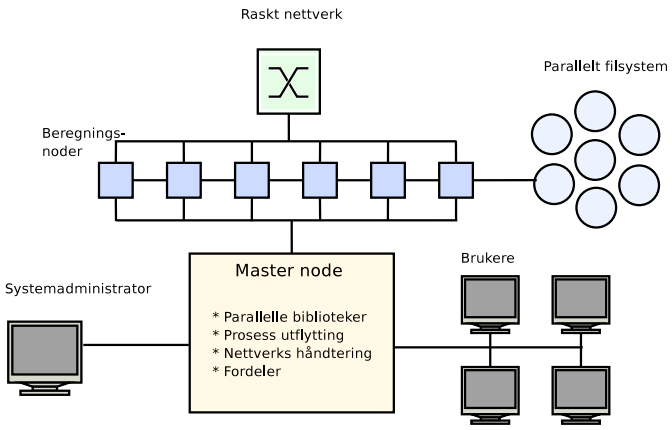
dynamikk, kjemi og geofysikk bruker gjerne Beowulf for å utføre beregningene raskt og kostnadseffektivt. Designeren av systemet kan øke ytelsen til clusteret proporsjonalt ved å legge til flere maskinenheter. En av hovedforskjellene på et Beowulf cluster og et cluster av arbeidsstasjoner er at Beowulf oppfører seg mer som en enkelt maskin. Nodene i clusteret er dedikert til clusteroppgavene, og de er vanligvis knyttet til verden utenfor gjennom kun en node.

Beowulf clustre utformes av standard maskinvare komponenter som er tilgjengelige på regulære utsalgssteder. Til dataoverføring og kommunikasjon mellom nodene benyttes meldingssending som Message Passing Interface og Parallell Virtual Machine. Effektiviteten til clusteret avhenger i stor grad av om programmet kan kjøres i parallell. Parallelle deler av programmet er de delene som kan utføres individuelt på separate prosesseringsenheter samtidig. En applikasjon uten parallelle deler vil utføres raskest på den raskeste noden i clusteret, uten andre noder involvert.

AMF (se kapittel 4.4) er et rammeverk for et distribuert system, så en parallel maskin som Beowulf er ikke intuitivt nyttig å bruke til dette formålet. Det kan allikevel være mulig å modifisere dette, eller bruke komponenter fra det, til å implementere rammeverket. I dette prosjektet er ikke en slik fremgangsmåte benyttet.

3.3.2.1 Scyld Beowulf

Scyld Beowulf tilbyr infrastrukturen som er nødvendig for å løse komplekse beregningsoppgaver på Linux clustre. Programvaren er neste generasjons arkitektur for Beowulf systemer, designet for enklere utvikling og administrasjon samt forbedret ytelse av clustrene. Arkitekturen til et Scyld Beowulf Cluster er gitt i figur 3.4.



Figur 3.4: Arkitekturen til et Scyld Beowulf Cluster

3.3.3 Spread

Problemet med å utvikle distribuerte systemer skyldes i stor grad behovet for å kommunisere mellom ulike komponenter i et system. Årsaken er at systemene benytter nettverk som ofte har en tendens til å feile. I et hvert distribuert system, for eksempel replikerte databaser eller et cluster av applikasjonsservere, er det visse usikkerhetsmomenter knyttet til tilstanden hos fjerntliggende komponenter. Grunnet kompleksiteten slike systemer innehar, er konstruksjonen av et pålitelig og effektivt distribuert system meget vanskelig.

Spread er et verktøy som tilbyr meldingstjenester med høy ytelse som tolererer feil både på interne og eksterne nettverk. Spread fungerer som en samlet meldingsbuss for distribuerte applikasjoner, og tilbyr multicast på applikasjonsnivå samt støtte for gruppekommunikasjon. Kommunikasjonskomponenten er designet for å innkapsle de krevende aspektene av asynkrone nettverk, samt å tillate konstruksjon av skalerbare distribuerte applikasjoner. Spread tilbyr en enkel API som lar en applikasjon bli medlem av en meldingsgruppe, sende melding til en gruppe, eller motta meldinger som er sendt til en gruppe fra andre medlemmer. Når en applikasjon sender ut en melding, sender APIet den til en Spread *deamon* som kjører på lokal maskin. Deretter distribueres meldingen til de andre maskinene i nettverket.

3.3.4 Transis

Transis [12, 13] er et multicast kommunikasjonslag som forenkler utviklingen av feiltolerante, distribuerte appliksjoner i et nettverk. Det støtter pålitelig gruppekommunikasjon for høytilgjengelige applikasjoner. Transis inneholder en protokoll for pålitelig meldingslevering som optimaliserer ytelsen for eksisterende maskinvare og tolererer partisjonering av nettverket.

Kommunikasjonssystemet støtter prosessgruppe kommunikasjon. En prosessgruppe er en gruppe av prosesser som identifiseres ved et navn valgt av brukeren. Meldinger adressert til gruppen vil da mottas av alle medlemmene. Ved bruk av en slik abstraksjon lettes brukeren fra jobben med å eksplisitt identifisere alle mottakerne av meldingene, samt å finne nettverksrutene til disse. Transis støtter også medlemskaps tjenester. Dersom prosesser legges til eller slettes fra grupper, for eksempel ved prosess krasj, endringer i nettverket eller brukerbestemte valg, rapporterer Transis endringen til alle aktive gruppemedlemmer mens konsistensen bevares mellom disse.

Transis bruker som nevnt multicast til å formidle meldinger i nettverket. En melding sendes da ut kun en gang, og når alle mottakerne i gruppen. Dersom TCP benyttes til gruppekommunikasjon på tilsvarende måte, må en socket opprettes mellom hvert par av prosesser. Meldinger må da sendes repeterende ganger over ulike kanaler for å nå frem til alle mottakerne. TCP krever også manuell håndtering av medlemskapsendringer. Brukeroppgavene blir dermed mer kompliserte. Transis rapporterer dette automatisk, og er således mer brukervennlig. I dette prosjektet er ikke Transis benyttet til gruppe kommunikasjon. Det er valgt å bruke TCP/IP, da dette ikke skaper noen vansker med bare to maskiner i clusteret (se kapittel 5.2).

3.3.5 Horus

Horus [14, 15] er et rammeverk for utvikling av distribuerte applikasjoner basert på gruppekommunikasjon, noe som kan være aktuelt i feiltolerante systemer, distribuerte systemer, gruppevare og applikasjoner som utnytter datareplikering. I Horus rammeverket er det utviklet en stor samling systemer og protokoller som tillater applikasjonsdesigneren å konstruere kommunikasjonsmoduler som eksakt møter applikasjonens krav til minimale kostnader.

Den grunnleggende ideen bak Horus er å tilby en arkitektur hvor protokollen som støtter gruppekommunikasjon kan endres under kjøring. Systemet søker således å møte kravene til applikasjonen og til omverdenen. Horus utfører dette ved å bruke et strukturert rammeverk for protokollkomposisjon. Systemet er skrevet i C, og har funksjonalitet for broadcasting, multicasting, pålitelig meldingssending og punkt til punkt kommunikasjon.

3.3.6 Heartbeat

Heartbeat er en del av Linux-HA prosjektet. Denne applikasjonen detekterer hvilke noder i clusteret som er oppe og nede, og oppdaterer deres status fortløpende. Heartbeat er brukt som komponent i dette prosjektet, og er nærmere beskrevet i kapittel 5.1.

Kapittel 4

Application Interface Specification

Application Interface Specification (AIS) er utviklet av Service Availability Forum (SA Forum). SA Forum er et samarbeid mellom industriledende kommunikasjons- og datafirmaer som ønsker å utvikle og publisere standarder for grensesnitt mot høytilgjengelig programvare [5]. Standardene skal støtte portabilitet av applikasjoner på tvers av høytilgjengelig mellomvare og maskinvare. Når disse kravene er oppfylt er det mulig for utviklere å skrive programvare som er portabel mellom implementasjoner fra ulike leverandører. I dette kapitlet beskrives SA Forum og deres definisjoner av ulike standarder, deriblant AIS. Deretter gis en oversikt over AIS standarden med vekt på *Cluster Membership Service*, som er den delen av AIS som vektlegges i implementasjonen av dette prosjektet.

4.1 SA Forum

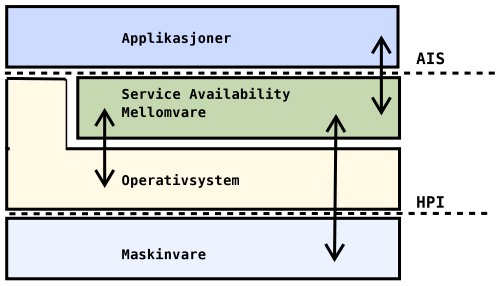
SA Forum utvikler standarder for å muliggjøre bruk av kontinuerlig tilgjengelige systemer som er uavhengig av maskinvare og mellomvare. Et av målene er at standard produkter som finnes på vanlige utsalgssteder skal kunne benyttes sammen med slike systemer. Det skal ikke være nødvendig med spesialtilpassede produkter eller enheter som er vanskelige å oppdrive.

Hovedmålet til SA Forum er å definere standard grensesnitt på en slik måte at utviklingen av høytilgjengelige systemer møter brukernes krav til pålitelighet og tilgjengelighet. Applikasjonene skal ikke skreddersys til hver enkelt plattform, men derimot fungere på alle mulige maskinvareplattformer.

4.1.1 Generell oversikt

Initielt vil SA Forum utvikle to grensesnitt spesifikasjoner [6]. Disse trenger ikke informasjon om verken CPU, operativsystem eller plattform. Spesifikasjonene

vil tilby muligheten til å integrere applikasjoner og systemer vertikalt, uten å tilpasse dette individuelt til spesielle plattformer. SA Forum definerer i den forbindelse et nytt programvarelag kalt *Service Availability Middelware*. Dette tilpasser seg grensesnittspesifikasjonene, og har ansvar for å håndtere redundans i maskinvaren samt å finne og reparere feil som oppstår. Redundans betyr her at det finnes ekstra maskinvareenheter i systemet. Dersom det oppstår en feilsituasjon på en komponent, håndteres dette ved at en annen komponent tar over førstnevntes prosesseringsoppgaver. Programvarelaget skal også tilby de tjenestene programvaren trenger for å reagere på systemfeil, uten tap av tjenestetilgjengelighet. Applikasjonene som kjører trenger da kun å kommunisere med dette mellomvarelaget. De blir fullstendig uavhengige av spesifikasjonene som finnes i den underliggende systemplattformen. Dette er en stor fordel da applikasjonene kan forbli uforandret uavhengig av hvilken plattform som benyttes.



Figur 4.1: Grensesnittene til *Service Availability* [6]

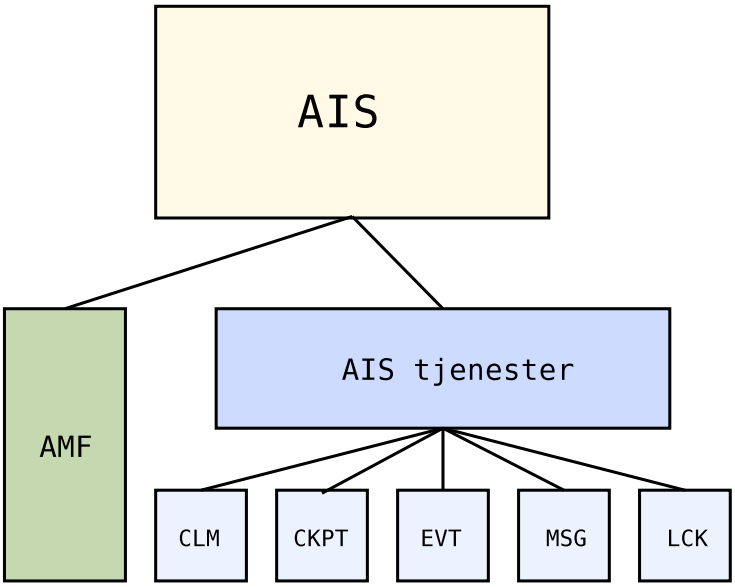
Her er AIS grensesnittet mellom applikasjonen og mellomvaren, det vil si biblioteket som implementerer mellomvaren. HPI (*Hardware Platform Interface*) er grensesnittet mellom mellomvaren og de ulike plattformkomponentene. I disse grensesnittene spesifiseres informasjonsflyten mellom programvareenhetene og deres semantikk.

HPI er en spesifikasjon for å håndtere en plattform som er uavhengig av spesifikk maskinvare. Når den er implementert, representerer HPI den fysiske maskinvaren og oversetter funksjonskall definert av spesifikasjonen til passende handling for maskinvaren. Dersom både AIS og HPI implementeres, er det mulig å lage systemer som tilbyr uavbrutte tjenester til brukerne.

AIS er en spesifikasjon for høytilgjengelig mellomvare som er uavhengig av firmaspesifikke implementasjoner. En åpen systemarkitektur i tillegg til en slik standard, tillater brukerne å velge blant ulike komponenter og kombinere disse etter eget behov. AIS støtter portabilitet av applikasjoner på tvers av høytilgjengelig mellomvare fra ulike tilbydere. Dermed kan brukeren velge den løsningen som er mest kostnadseffektiv i stedet for å binde seg til en spesiell maskin- og programvare. Muligheten til å variere utvalget i bruk av både programvare, maskinvare, operativsystem og applikasjoner gir større fleksibilitet i design av systemet. Brukeren kan enkelt erstatte enkeltkomponenter, og dermed tilpasse og optimalisere systemet etter behov.

4.2 AIS

AIS består av *Availability Management Framework* (AMF) og fem grupper av AIS-tjenester. Disse inkluderer *Cluster Membership Service* (CLM), *Checkpoint Service* (CKPT), *Event Service* (EVT), *Message Service* (MSG) og *Lock Service* (LCK). AIS-standarden er beskrevet i et eget dokument bestående av over 300 sider [3], og inneholder system- og grensesnittbeskrivelsene av AMF og de ulike AIS-tjenestene. Strukturen til AIS er gitt i figur 4.2.



Figur 4.2: Strukturen til AIS

4.3 De ulike tjenestene i AIS

AIS tilbyr som nevnt fem ulike tjenester som utgjør hovedfunksjonaliteten til clusteret. Her følger en kort beskrivelse av hver av disse [4].

4.3.1 Cluster Membership Service

Denne tjenesten sørger for at applikasjonene får medlemskapsinformasjon om nodene i clusteret. Dette er kjernen i alle clustrede systemer. Et cluster består av en mengde noder hvor alle nodene har unike identifikatorer. Etter hvert som nodene kommer til og forlater clusteret, oppdateres tjenesten med ny informasjon slik at den alltid gjenspeiler nåværende tilstand. Via ulike funksjoner kan denne informasjonen hentes ut av prosessene. Disse kan også registrere en callback funksjon hos *Cluster Membership Service*, og motta meldinger hver gang tilstanden i clusteret endres.

4.3.2 Checkpoint Service

Checkpoint Service lar prosessene registrere checkpoint data forløpende. Det blir da enklere å implementere en feiltolerant applikasjon. I et checkpoint lagres en kopi av dataene med jevne mellomrom. Dette ligger typisk i minnet og ikke på disk, slik at dataene skal være raskt tilgjengelige. Når AMF gjenoppretter en prosess som har feilet, brukes *Checkpoint Service* for å finne tilbake til forrige checkpoint. Utførelsen kan deretter fortsette fra dette punktet med de lagrede dataene som hentes herfra, da disse er fra en tilstand før feilen opptrådte. En slik prosedyre minimerer omfanget av eventelle feil som måtte oppstå. Et gitt checkpoint kan ha tilhørende data lagret på flere noder i clusteret for å sikre seg mot nodefeil. For å unngå opphoping av slike data, slettes de av tjenesten dersom de har vært inaktive en gitt tidsperiode. *Checkpoint Service* er nærmere beskrevet i kapittel 4.5.

4.3.3 Event Service

Event Service er en *publish/subscribe* (utgiver/abonnement) kommunikasjonsmekanisme. Utgiveren kommuniserer med en eller flere abonnementer over en hendelseskanal. Flere utgivere og abonnenter kan kommunisere over samme kanal. Kanalene er globale i hele clusteret, og identifiseres ved et unikt navn. Abonnentene er anonyme, noe som medfører at de kan slutte seg til og forlate hendelseskanaler uten å involvere utgiverne.

Hendelser består av en standard *header* med attributter, og null eller flere bytes med hendelsesdata fra utgiveren. For å bruke *Event Service* må prosessen opprette en hendelseskanal. Deretter åpnes denne slik at prosessen kan opptre som både utgiver og abonnent over kanalen.

4.3.4 Message Service

Message Service tilbyr tjenester som gjør at prosesser på ulike noder, eller innad i samme node, kan kommunisere. Meldingene skrives til og leses fra meldingskøer. Dersom en prosess feiler kan standbyprosesser overta prosesseringen av meldingen fra køen. Prosessen som har sendt meldingen kommuniserer bare med køen, og vet ingenting om en eventuell feil som har oppstått. Køer kan grupperes sammen, og hver gruppe får da sitt unike navn. Prosessen forholder seg kun til dette navnet, og har ikke behov for kjennskap til antall køer eller deres lokasjoner.

4.3.5 Lock Service

Lock Service er en distribuert tjeneste, beregnet på bruk i et cluster, hvor prosessene i ulike noder konkurrerer med hverandre om å få tilgang til en delt ressurs. Det tas da i bruk låser for å synkronisere aksessen til disse dataene mellom prosessene. Dersom en prosess bruker en ressurs, settes en lås slik at andre prosesser ikke kan aksessere ressursen samtidig og dermed skape en konflikt.

4.4 AIS Availability Management Framework

Availability Management Framework (AMF), også kalt rammeverket, er en programvareenhet som tilbyr tjenestetilgjengelighet. AMF koordinerer redundante ressurser innad i et cluster til å levere et system som maskerer enkeltfeil. Rammeverket tilbyr en rekke funksjoner for å muliggjøre høytilgjengelige applikasjoner. Den bestemmer tilstanden til en komponent (om den er oppe eller nede) ved å anrope callbackfunksjonene til komponenten. Komponenter kan også få informasjon fra AMF om andre komponenters tilstand. For en utfyllende beskrivelse av denne enheten henvises det til [3] og [4].

4.5 Checkpoint Service

Checkpoint Service gir prosesser muligheten til å lagre checkpoint data inkrementelt. Ved å bruke denne tjenesten til å lagre checkpoint data beskyttes applikasjonen mot feil. Når AMF skal utføre *recovery* etter en feilsituasjon, enten ved bruk av restart eller en *failover* prosedyre, brukes *Checkpoint Service* for å finne tilbake til forrige checkpoint og gjenoppta utførelsen herfra med de lagrede dataene. Denne tilstanden vil være fra før feilen oppsto, og en slik fremgangsmåte vil dermed minimere feilomfanget.

Checkpoints er entiteter som gjelder for hele clusteret, og de er tilordnet unike navn. En kopi av dataene lagret i checkpointet kalles et replikat, og er typisk lagret i clusternodenes minne fremfor på disk. Dette er av ytelsesmessige årsaker, da data lagret på disk vil ta lenger tid å aksessere. Dataene fra ett checkpoint lagres på flere clusternoder for å beskytte mot nodefeil. Dersom en node går ned, vil checkpoint dataene som var lagret her allikevel kunne hentes frem fra andre noder. En slik fremgangsmåte vil derimot ikke beskytte mot en komplett cluster nedstengelse så lenge dataene bare lagres i minnet.

4.5.1 Modellen for Checkpoint Service

Checkpoints

Tjenesten *Checkpoint Service* håndterer som nevnt en mengde entiteter kalt checkpoints, som prosesser bruker for å lagre sin tilstand. En gitt prosess kan benytte ett eller flere checkpoints til dette. Checkpoints kan opprettes, lukkes og slettes dynamisk ved hjelp av funksjonskall. Når et checkpoint har blitt slettet kan det ikke lenger aksesseres via det globale navnet det er tilordnet, men prosesser som allerede har åpnet checkpointet kan fortsette å aksessere dette til det lukkes. Dette betyr at dersom et checkpoint slettes vil ressursene assosiert med dette først frigjøres når checkpointet lukkes av den siste prosessen som har dette åpent.

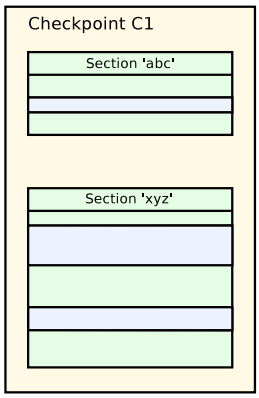
For å unngå akkumulering av ubrukte checkpoint i systemet, innføres det en oppbevaringstid. Når et checkpoint ikke har vært åpnet av noen prosesser i løpet av denne tidsperioden, slettes det automatisk av *Checkpoint Service*. Dersom en prosess avslutter på en unormal måte, lukker *Checkpoint Service* alle åpne checkpoint assosiert med denne.

Sections

Hvert checkpoint er strukturert som en mengde sections som kan opprettes og slettes dynamisk av prosessene. Maksimalt antall sections defineres ved opprettelse av checkpointet. Sections som tilhører et checkpoint kan dynamisk opprettes eller slettes så lenge det totale antallet ikke overstiger verdien for maksimalt antall. Innad i et checkpoint er hver section identifisert ved en unik identifikator. Sections er kun unike innad i et checkpoint. Section identifikatorer kan spesifiseres når man oppretter en prosess, eller de kan allokeres dynamisk av *Checkpoint Service*.

Det kan eksistere flere sections av ulike størrelser i et checkpoint samtidig, men en maksimal størrelse på disse spesifiseres ved opprettelse av checkpointet. Størrelsen kan endres dynamisk ved kall til spesifikke metoder. Sections inneholder rådata, og disse krypteres ikke av *Checkpoint Service.* Det er prosessenes ansvar å kryptere innholdet dersom det skal tas hensyn til heterogenitet.

Secions har en utløpstid. Dersom denne nås, slettes sectionen automatisk av *Checkpoint Service* - uavhengig av om noen prosesser har checkpointet åpent.



Figur 4.3: Et checkpoint kan bestå av flere sections av ulik størrelse

Checkpoint replikat

En kopi av dataene lagret i et checkpoint kalles et checkpoint replikat, eller bare et replikat. Det kan maksimalt eksistere ett replikat for et spesifikt checkpoint på en gitt node. Et checkpoint kan ha flere replikater implementert på ulike noder. Dersom et replikat er lokalisert på noden hvor checkpointet er åpnet, kalles det et lokalt replikat. Håndtering av replikatene er transparent for applikasjons programmereren.

Checkpoint data aksess

En prosess kan bruke *handle*n returnert av *Checkpoint Service* ved initialiseringen av et checkpoint til å utføre lese- og skriveoperasjoner på checkpointet. En

enkelt operasjon kan aksessere forskjellige deler av ulike seksjoner i checkpointet samtidig. Kravene angående konsistens på ulike replikater assosiert med et gitt checkpoint kan ha negativ effekt på ytelsen til skriveoperasjonene. For å tillate fleksibilitet for programmererne av *Checkpoint Service* er ikke sterk atomitet og streng ordning på semantikken et krav. Det vil si:

- Dersom en feil oppstår under en skriveoperasjon er det ikke et krav at alle replikater er identiske. Noen replikater kan ha blitt modifisert av operasjonen, andre ikke.
- Dersom to prosesser utfører en skriveoperasjon samtidig til samme partisjon av checkpointet, er ingen global ordning av replikatoppdateringene sikret. Etter at begge skriveoperasjonene er utført, kan noen replikater inneholde data skrevet av en prosess mens andre replikater inneholder data skrevet av en annen prosess. Det er prosessenes ansvar å bruke fornuftige synkroniseringsmekanismer (som for eksempel *Lock Service*, se kapittel 4.3.5) hvis det er krav om en slik global oppdatering.

For å imøtekomme ulike avveiinger mellom ytelsen på oppdatering av checkpoint og konsistens på replikater, tilbys ulike valgmuligheter ved opprettelsen av et checkpoint. Disse er beskrevet i påfølgende avsnitt.

Synkron oppdatering

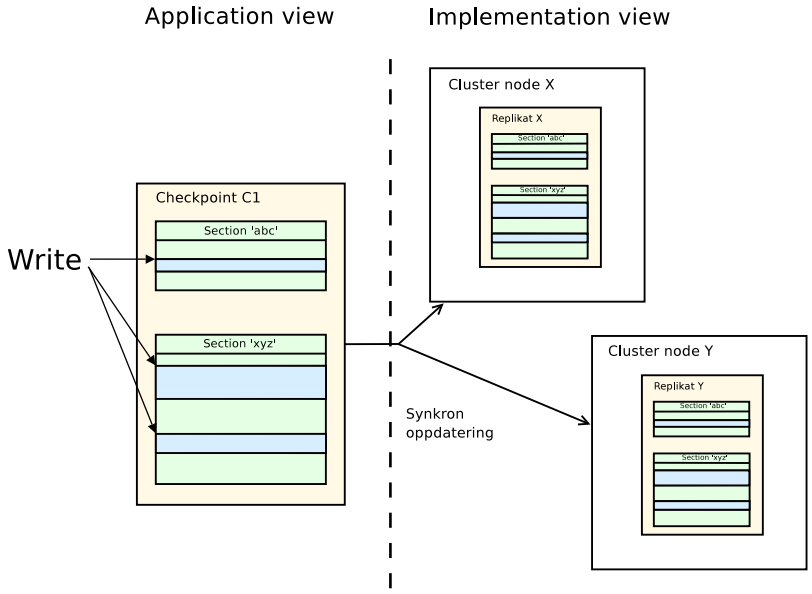
Når et checkpoint opprettes med valget synkron oppdatering, vil kall til skriving, oppdatering og sletting av en section kun returnere når alle checkpoint replikater er oppdatert (se figur 4.4). I tillegg garanterer *Checkpoint Service* at det ikke finnes partielle oppdateringer i replikatet. Enten er det oppdatert med alle dataene spesifisert i kallet, ellers oppdateres det ikke i det hele tatt. *Checkpoint Service* garanterer altså at alle replikatene opprettet med synkron oppdatering er identiske. Det er ikke spesifisert hvilket replikat checkpoint dataene leses fra. Et slikt checkpoint lages ved å spesifisere flagget SA_CKPT_WR_ALL_REPLICAS ved opprettelse.

Asynkron oppdatering

Når et checkpoint er opprettet med valget asynkron oppdatering, returnerer kall til skriving, oppretting og sletting av en section med en gang det aktive replikatet er oppdatert. Andre replikater oppdateres asynkront (se figur 4.5). Checkpoint replikatet som oppdateres synkront kalles det aktive replikatet. For å garantere at en prosess ikke leser data som er foreldet, leser *Checkpoint Service* alltid fra det aktive replikatet.

Checkpoint Service garanterer ikke at alle replikater opprettet med asynkron oppdatering er identiske. En prosess kan derimot sikre at *Checkpoint Service* synkroniserer alle checkpoint replikater ved å bruke et funksjonskall for å sende ut checkpoint data til alle replikater.

Det er to varianter av checkpoints innenfor asynkron oppdatering. For den første garanterer *Checkpoint Service* atomitet når replikater oppdateres. Det vil si at et



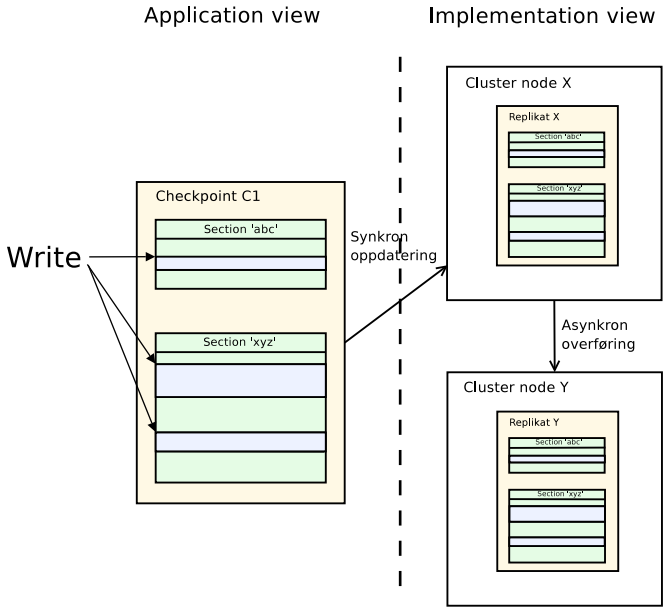
Figur 4.4: Synkron skriving

replikat enten er oppdatert med alle dataene spesifisert i skrivekallet, eller ikke oppdatert i det hele tatt. Disse lages ved å spesifisere flagget SA_CKPT_WR_ACTIVE_REPLICA ved opprettelse. Den andre varianten kalles partiell oppdatering. Denne garanterer ikke atomitet, men markerer sections modifisert av skrivekall som korrupte dersom det oppstår en feil under oppdatering av replikatet. Korrupte sections kan ikke aksesseres ved bruk av vanlige lese- og skrivekall - de kan kun overskrives eller slettes. Slike checkpoints lages ved å spesifisere flagget SA_CKPT_WR_ACTIVE_REPLICA_WEAK ved opprettelse. Målgruppen for dette valget er de applikasjonene som ikke ønsker å betale den prisen for ytelsen assosiert med beskyttelsen mot partielle oppdateringer.

Håndtering av replikater for sammenstilte og ikke-sammenstilte checkpoints

- Sammenstilte checkpoints

  Når et checkpoint er brukt i asynkron modus, oppnås optimal ytelse for oppdatering av et checkpoint når det aktive replikatet er lokalisert på samme node som prosessen som aksesserer checkpointet. Fordi prosessen som aksesserer checkpointet kan endres avhengig av hvilken rolle den tilordnes av AMF, vil optimal ytelse imidlertid kun oppnås dersom applikasjonen informerer *Checkpoint Service* om hvilket replikat som skal være aktivt til et spesielt tidspunkt. Dette kan gjøres for checkpoints som innehar sammenstillingsattributtet. Disse kalles sammenstilte checkpoints. Når slike checkpoints opprettes eksisterer det ingen aktive replikater før et lokalt replikat er satt til å være aktivt via et funksjonskall. Et aktivt replikat forblir aktivt til brukeren eksplisitt setter et annet replikat til å



Figur 4.5: Asynkron skriving

ha denne statusen, eller til replikatet ødelegges (for eksempel ved at noden det ligger på krasjer). I det siste tilfellet vil det ikke eksistere noen aktive replikat før brukeren fysisk setter et nytt.

Når et nytt replikat blir satt til å være aktivt, garanterer *Checkpoint Service* at dette er fullstendig synkronisert med det forrige aktive replikatet. Datakonsistens for lesing, skriving og skriverekkefølge av replikater ivaretas som om endringen av det aktive replikatet aldri fant sted. Lesing og skriving kan blokkeres til synkroniseringen er utført.

- Ikke-sammenstilte checkpoints

  Checkpoints som opprettes uten sammenstillingsattributtet kalles ikke-sammenstilte checkpoints. Håndtering av deres replikater, samt bestemmelsen om de settes til aktive eller ikke, utføres av *Checkpoint Service*. Prosessene som benytter seg av denne tjenesten har ingen informasjon om lokasjonene til de aktive replikatene. *Checkpoint Service* kan opprette replikater i tillegg til de som ble laget ved opprettelse av checkpointet. Dette kan være nyttig for å øke tilgjengeligheten til checkpointene ytterligere. Hvis det for eksempel eksisterer to replikater på et gitt tidspunkt, og noden det ene er lokalisert på blir satt til OUT_OF_SERVICE, kan *Checkpoint Service* allokere et annet replikat på en annen node mens den opprinnelige noden er utilgjengelig.

Persistens av checkpoints

Checkpoint Service lagrer checkpoint dataene i minnet til cluster nodene. Dersom tjenesten stopper å kjøre på alle nodene som har et replikat for dette check-

pointet, vil ikke checkpointet og dets sections overleve selv om oppbevaringstiden ikke har utløpt. En slik situasjon kan forårsakes av administrasjonsutførelser eller nodefeil.

Kapittel 5

Teknologier

I dette kapitlet introduseres de viktigste teknologiene som er benyttet i prosjektet. Første delkapittel gir en beskrivelse av clusterprogramvaren Heartbeat. Deretter følger beskrivelser av hovedtrekkene ved TCP/IP og UDP.

5.1 Heartbeat

I den implementerte løsningen er det valgt å benytte clusterprogramvaren Heartbeat, utviklet av Linux-HA, til deteksjon av feil på noder. Dette valget ble tatt fordi Heartbeat er vel utprøvd på andre systemer, og det er godt dokumentert at programvaren fungerer på en tilfredsstillende måte for våre formål. Da Heartbeat er mye brukt i andre prosjekter, finnes det diverse internettbaserte diskusjonsforum hvor man kan få svar på spørsmål angående problemstillinger som oppstår underveis. Alle disse elementene talte for at Heartbeat ville være en komponent som var godt egnet for å administrere clusteret i dette prosjektet.

Linux-HA er et *open source*-prosjekt som jobber med å tilby en høytilgjengelig clusterløsning for Linuxplattformen [17]. Heartbeat er en av de viktigste bestanddelene i prosjektet. Programmet er kompatibelt med alle kjente Linuxplattformer, i tillegg til Free BSD og Solaris. Heartbeat komponentene i Linux-HA er blant komponentene på lavest nivå i systemet. Hensikten med disse er at påliteligheten skal være meget høy. Det er derfor viktig at komponentene er enkle å forstå, enkle å debugge og meget robuste. De er designet slik at de skal kunne kjøre kontinuerlig i mange år uten at minnelekkasje eller andre feil oppstår.

Ethvert høytilgjengelig system er avhengig av to grunnleggende tjenester for å fungere: å bli informert om hvilke cluster medlemmer som blir med i og forlater clusteret, og å tilby kommunikasjonstjenester for å håndtere clusteret [18]. For å innhente informasjon om medlemsskap i et cluster sjekkes det om det er mulig å kommunisere med nodene. Dersom kommunikasjon kan opprettes, regnes noden som medlem av clusteret. Dersom det ikke er mulig å opprette kommunikasjon med noden, regnes den som død. For å avgjøre slike situasjoner i dette prosjektet er altså Heartbeat benyttet. Hver maskin som er en del av clusteret genererer *I'm-alive*-meldinger som sendes til andre noder i clusteret.

Dersom et gitt antall slike meldinger fra en node har uteblitt, regnes noden som død eller borte fra clusteret. Med Heartbeat sendes *I'm-alive*-meldinger ut i et alle-til-alle scenario på periodisk basis. For store clustre kan det være en fordel å bruke ulike muliticast eller broadcast teknikker for å unngå $O(n^2)$ operasjoner som krever mye ressurser.

Nodene i clusteret må kommunisere for å kunne utføre operasjoner. Dersom kommunikasjonen ikke er pålitelig, vil heller ikke clusteret være pålitelig. Siden cluster håndtering typisk er transaksjonsorientert, med transaksjoner spredt over hele clusteret, er det vanlig å kommunisere med alle nodene samtidig. Det kan for eksempel oppstå en situasjon hvor en av nodene registrerer en hendelse som skal medføre en tilstandsendring for hele clusteret. Denne noden må i såfall formidle beskjeden til alle medlemmer av clusteret. Noden sender da typisk ut en melding til alle medlemmene, og venter på bekreftelse fra disse.

Vanligvis er det ønskelig å detektere nodefeil i løpet av sekunder. Dette betyr i praksis at heartbeat-meldingene som sendes mellom nodene må leveres i løpet av mye kortere tid enn dette. Det er også ønskelig å vite om backup-linkene til systemet fungerer, og rapportere om utfallet. Dette reduserer sjansen for multiple feil som systemet ikke kan maskere. Dersom backup kommunikasjonskabelen ikke fungerer og hovedkabelen blir ødelagt, vil ikke systemet være i stand til å kommunisere. Liknende situasjoner kan unngås dersom det rapporteres om feil i backupløsninger.

På bakgrunn av disse observasjonene kom utviklerne av Heartbeat frem til designet av programmet. Heartbeat tilbyr de grunnleggende funksjonene som kreves av et høytilgjengelig system. Dette innebærer starting og stopping av ressurser, overvåking av tilgjengeligheten til systemene i clusteret og overføring av eierskap til delte IP-adresser mellom noder i clusteret. Heartbeat protokollen har følgende karakteristika:

- Mulighet for multicast
- Garantert pakkelevering
- Ikke garantert pakkerekkefølge
- Flyt- og køkontroll er ikke inkludert

De to første punktene er nødvendige av årsaker som ligger i høytilgjengelighetens natur. Rekkefølgen på pakkene trenger ikke å garanteres. Dette er på grunn av den strenge forespørsel/svar-implementasjonen som ligger i cluster-håndteringsfunksjonene som plasseres i et høyere lag i Heartbeat applikasjonen. Dette utelater også nødvendigheten av flyt- og køkontroll, siden det ikke vil sendes flere pakker før svar vedrørende tidligere sendte pakker er mottatt.

Når det gjelder design av multicast protokoller finnes to hovedprinsipper:

- Sender-initiert
- Mottaker-initiert

I sender-initierte multicastprotokoller sender mottakeren bekreftelse på hver av pakkene han mottar. Senderen vedlikeholder en timer og sender pakkene på nytt dersom de ikke er mottatt innen et gitt tidsintervall. I mottaker-initiert muliticast er mottakeren ansvarlig for å detektere pakketap. Dette gjøres ved hjelp av sekvensnummer. Når mottaker detekterer at en pakke har gått tapt, etterspør han pakken på nytt. Det er en variant av dette som brukes i Heartbeat. Hver mottaker etterspør en ny pakke maks én gang per sekund, og hver sender vil på nytt sende (ved bruk av cluster broadcast) hver pakke maks én gang per sekund.

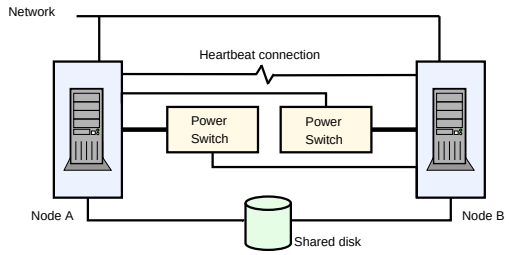
Siden clustermedlemmer må kunne stole fullt og helt på hverandre, må kommunikasjonssystemet enten ha fysisk sikker kommunikasjon eller kommunikasjon hvor det ikke er mulig å oppgi falsk IP-adresse. Det kan forekomme at en node ber en annen om å stoppe betjeningen av en gitt IP-adresse, stenge ned alle programmer, restarte eller utføre andre operasjoner med alvorlige konsekvenser. En clusterkommunikasjonskanal kan være en måte for inntrengere å bryte seg inn i maskinene på, og dette er noe man ønsker å unngå. I Heartbeat løses problemet ved digital signering av hver pakke, og kun godkjenning av pakker med riktig signatur.

Et høytilgjengelig system skal ideelt sett være operativt uten avbrytelse i årevis, og overleve både programvare- og maskinvare oppdateringer. Systemet må derfor tolerere oppgraderinger i fart, det vil si mens det utfører tjenester. Det er derfor nødvendig at gamle versjoner av programvaren aksepterer meldinger fra nyere versjoner, og ignorerer eventuelle felt i meldingen de ikke forstår.

Det finnes en API for Heartbeat som muliggjør kommunikasjon på tvers av lagene i clusteret. Her kan man, blant flere funksjoner, finne status på noder og linker, observere endringer i tilstanden samt sende og motta pålitelige meldinger til og fra andre noder.

Når to systemer kan aksessere samme disk, er det nødvendig å unngå samtidig skriving til en enkelt partisjon for å ivareta dataintegriteten. Vanligvis er dette noe som ikke kan oppstå. Dersom en server feiler, vil den andre ta over og starte skrivingen fra stedet der den første slapp. Uheldigvis er det slik at hvis en node tror den andre er nede, selv om den faktisk ikke er det, kan man risikere at begge prøver å aksessere det delte mediumet. Situasjonen hvor to ulike systemer begge feilaktig tror det andre er nede, kalles et partisjonert cluster. Dersom dette forekommer, og en delt disk er involvert, kan dataene og filsystemet ødelegges. Heartbeat benytter en maskinvareenhet som kalles STONITH for å sikre at en antatt død server ikke interfererer med nåværende cluster operasjoner, og at den ikke ødelegger delte disker. STONITH står for "Shoot The Other Node In The Head" ("Skyt den andre noden i hodet"), og er obligatorisk å bruke dersom man har delte disker. Når clusterprogramvaren antar at en node har krasjet, brukes STONITH til å stenge ned strømtilførselen til noden. Den vil da ikke ha noen mulighet for å skrive til disk. Etter et forhåndsdefinert intervall vil enheten koble strømmen til igjen og la Linux-systemet starte opp og muligens reintegrere noden i clusteret. STONITH er en teknikk som sikrer at selv om høytilgjengelighet er ønskelig, er dataintegritet en enda høyere prioritet. En to-node konfigurasjon med STONITH er illustrert i figur 5.1.

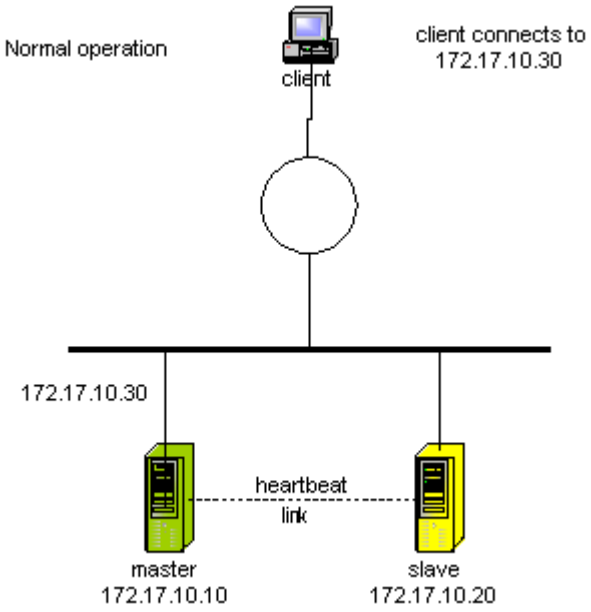
Dersom STONITH ikke brukes, kan feilkonfigurering eller programvarefeil la serveren tro at den andre siden er nede selv om den ikke er det. Dette kalles en



Figur 5.1: En to-node konfigurasjon med STONITH

"split brain" tilstand.

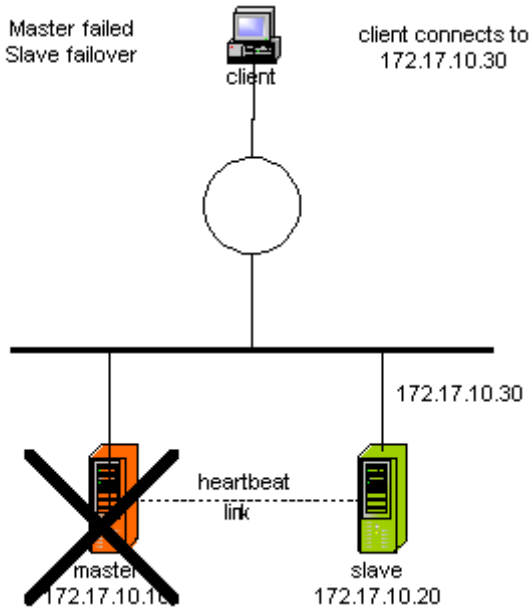
I dette prosjektet benyttes det et cluster bestående av to noder, hvor en er primær og en er standby. Primæren starter opp applikasjonen, og kjører helt til det eventuelt oppstår en feil. Standbynoden vil da ta over dens oppgaver, og fortsette kjøringen av applikasjonen fra der primær noden krasjet. Clusteret har en statisk, virtuell IP-adresse som brukes av klienter som skal kommunisere med clusteret. Denne IP-adressen vil hele tiden håndteres av den noden som er operativ. Under normal kjøring er det primærnoden som eier ressursene og IP-adressen (se figur 5.2).



Figur 5.2: Normal kjøring med to noder [20]

I et *failover* scenario vil standbynoden overta primærnodens ressurser og den virtuelle IP-adressen, og starte primærnodens prosesser. Klienter vil nå kommunisere med standbynoden ved å bruke samme IP-adresse som før (se figur

5.3). De vil derfor ikke merke at det har skjedd en endring i hvilken node de kommuniserer med, da nodene utfører samme oppgaver.



Figur 5.3: Primær noden går ned, og failover oppstår

5.2 TCP/IP og socketkommunikasjon

TCP/IP står for Transmission Control Protocol / Internet Protocol, og er en protokoll for overføring av data. I denne implementasjonen er TCP/IP benyttet sammen med socketkommunikasjon. Sockets tilbyr en pålitelig form for kommunikasjon, samtidig som kommunikasjonen kan finne sted både mellom prosesser lokalt og over et nettverk [21].

Kommunikasjonsprotokollen TCP/IP er satt sammen av tre lag:

- TCP

  TCP er ansvarlig for å verifisere korrekt overføring av data fra klient til server. TCP-laget på avsendermaskinen formaterer om meldingene til pakker. Disse sendes via nettverket til mottakermaskinen, hvor de pakkes ut av TCP-laget og rekonstrueres til den opprinnelige meldingen. Protokollen sikrer at dataene ankommer sekvensielt og feilfritt. Dersom dataene under overføring blir skadet, tapt eller duplisert, eller at rekkefølgen blir endret, håndteres dette av TCP. Den sørger for at rekkefølgen ved ankomst er den samme som ved sending. Dette er viktig når funksjonskall med tilhørende parameterlister skal overføres mellom prosessene.

- IP

  IP er ansvarlig for å overføre datapakkene til korrekt mottakermaskin. Pakkene har en 4 bytes lang destinasjonsadresse (IP nummeret) som angir hvilken node den skal ende opp hos. Hver *gateway* i nettverket sjekker destinasjonsadressen, og sender pakken videre basert på denne informasjonen. Selv om sammenhørende pakker fra samme avsender rutes ulike veier, settes de sammen i riktig rekkefølge ved destinasjonen.

- Sockets

  Sockets er betegnelsen på en mengde subrutiner som tilbyr et veldefinert grensesnitt slik at bruker-/applikasjonsprosesser kan aksessere TCP/IP. Grensesnittet består av et antall veldefinerte kall som blant annet gjør det mulig å åpne og lukke forbindelser, og å sende og motta data. Stream sockets er pålitelige toveis forbindelser for kommunikasjon. De tilbyr et høyt nivå av overføringskvalitet, som i hovedsak skyldes bruken av TCP.

5.3 UDP

UDP står for User Datagram Protocol, og er en kommunikasjonsprotokoll som tilbyr et begrenset antall tjenester når meldinger utveksles mellom maskiner i et nettverk som bruker IP (Internet Protocol) [22]. UDP er et alternativ til TCP, og sammen med IP kalles den også UDP/IP. Som TCP bruker UDP Internet Protocol til overføring av data mellom maskiner. Meldingsenhetene som overføres kalles datagrammer. I motsetning til TCP tilbyr ikke UDP tjenesten med å dele meldingen opp i pakker for så å sette de sammen igjen til den opprinnelige meldingen hos mottaker. Det sikres ikke at dataene ankommer sekvensielt. Dette betyr at applikasjonenen som anvender UDP må sørge for at hele meldingen har ankommet mottakeren, og at rekkefølgen på dataene er korrekt. Siden UDP er en mye enklere protokoll enn TCP inneholder meldingsheaderen færre *bytes* og oppnår derfor mindre nettverksoverhead.

Kapittel 6

Implementasjon

I dette kapitlet beskrives implementeringen som er utført i prosjektet. I første underkapittel introduseres det som karakteriserer løsningen på et overordnet plan, samt begrensningene ved implementasjonen. Andre underkapittel gir en beskrivelse av hvordan kommunikasjonen mellom ulike komponenter i løsningen er realisert. Tredje underkapittel beskriver det utvalget av CKPT funksjoner som inngår i prosjektet, og forklarer hvordan disse samt de relevante datastrukturene er implementert, . Siste underkapittel gir en mer detaljert gjennomgang av programflyten i den implementerte løsningen. Det legges i tillegg vekt på å gi et inntrykk av hvilken rolle de implementerte CKPT funksjonene spiller.

6.1 Overordnet beskrivelse av løsningen

Løsningen som er utviklet baserer seg på en videreutvikling av den implementeringen som ble gjort i [4]. Her ble deler av det grunnleggende rammeverket, AMF, realisert i programmeringsspråket C, som også er det språket selve AIS standarden spesifiseres i. Som utviklingsmiljø benyttet man her instituttets Solaris-maskiner. For videreføringen av prosjektet ble det derimot tatt i bruk en lokal installasjon av Debian Linux på egne arbeidsstasjoner. Dette skyldtes i hovedsak behovet for å kunne installere spesifikk programvare på de aktuelle maskinene.

Løsningen som ble implementert i [4], som i sin helhet kjørte på én node, er i dette prosjektet utvidet til å omfatte et cluster bestående av to noder. Disse nodene benytter en cluster programvare, kalt Heartbeat (se kapittel 5.1), for å utveksle I'm-alive-meldinger (heartbeats) seg imellom. Ved hjelp av denne teknologien har hver av de to nodene mulighet til å detektere krasj hos den andre noden. Gjeldende versjon av Heartbeat tillater ikke at mer enn to noder inngår i clusteret, og løsningen det eksperimenteres med i dette prosjektet er derfor også begrenset til å omfatte kun to noder. Dersom flere noder skulle inngått i clusteret måtte det bli tatt i bruk metoder for å garantere konsistens over alle nodene, for eksempel 2-fase commit. Det er ikke tatt høyde for en slik problemstilling i dette prosjektet.

Hver av nodene i clusteret kjører to prosesser: én hovedprosess som emulerer AIS og én prosess som representerer en feiltolerant brukerapplikasjon. Denne applikasjonen benytter funksjoner både fra AIS-biblioteket og fra andre C-biblioteker som lenkes inn. Gjennom kall til AIS opprettes en forbindelse mellom brukerapplikasjon og hovedprosess som åpner for toveis kommunikasjon. Likeledes vil en forbindelse mellom de to hovedprosessene opprettes, slik at disse kan utveksle checkpointmeldinger. Kommunikasjonen mellom løsningens ulike komponeneter er illustrert i figur 6.1.

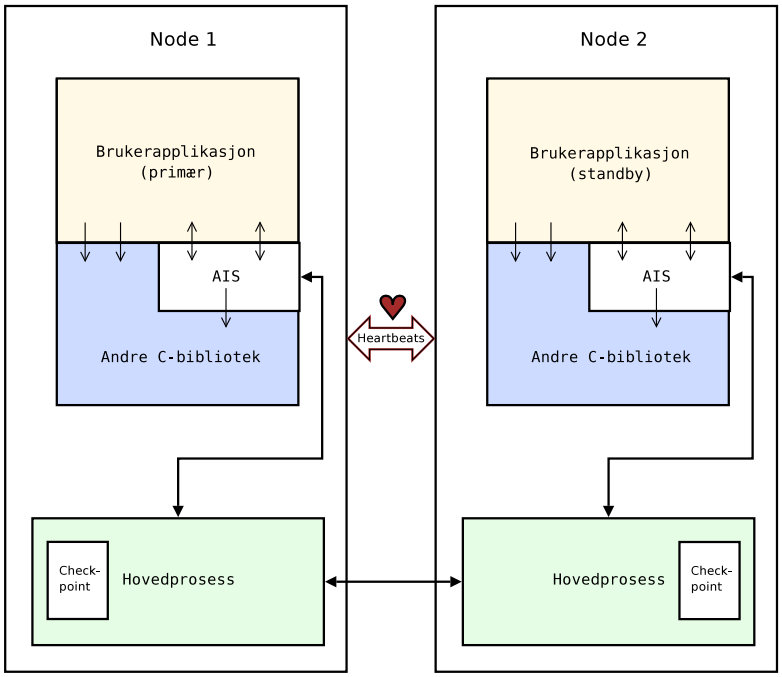


Figure 6.1: Kommunikasjon mellom ulike komponenter som inngår i løsningen

Ved oppstart vil den ene noden ha status som primær mens den andre opptrer som standby. Primærnodens brukerapplikasjon vil etter oppstart begynne å utføre sin tjeneste, mens tilsvarende applikasjon på standbynoden, etter å ha initialisert biblioteket, vil vente i denne starttilstanden inntil den får beskjed om at primærnoden har gått ned. Når dette inntreffer, vil noden som i utgangspunktet var standby skifte status til primær og gi sin brukerappliksjon beskjed om å starte normal utførelse.

Som testformål er det utviklet en enkel brukerapplikasjon hvis oppgave er å skrive ut en teller som inkrementeres hvert sekund. For at standbyapplikasjonen ved takeover skal ha mulighet til å finne tilbake til den tilstanden primærapplikasjonen var i da den krasjet, er det helt nødvendig at primærapplikasjonen underveis checkpointer sin tilstand over til standbynoden. Dette gjøres i form av såkalt State Checkpointing (se kapittel 2.3.2). For det enkle tellerprogrammet representeres tilstanden som må overføres kun ved et tall, men for mer avanserte brukerapplikasjoner kan informasjonen som må checkpointes være svært omfattende. For eksempel vil representasjonen av tilstanden til en mySQL-database

være betydelig mer kompleks.

I AIS standarden er det tilrettelagt for to former for skriving av checkpoint; synkron og asynkron oppdatering, se kapittel 4.5.1. Den implementerte løsningen omfatter imidlertid kun synkron skriving, der checkpointet skal være reflektert i minnet på begge noder før skriveoperasjonen anses som fullført. I tillegg sørger AIS for at det ikke forekommer noen delvis oppdatering av checkpointdataene; enten oppdateres checkpointet i sin helhet, ellers oppdateres det ikke i det hele tatt.

Det er ikke foretatt en fullstendig implementering av CKPT-funksjonene som er angitt i AIS-standarden. De aktuelle funksjonene som er utviklet og benyttet i dette prosjektet er beskrevet i kapittel 6.3.1.

6.2 Kommunikasjon

Som illustrert i figur 6.1 foregår kommunikasjonen i den implementerte løsningen på tre plan:

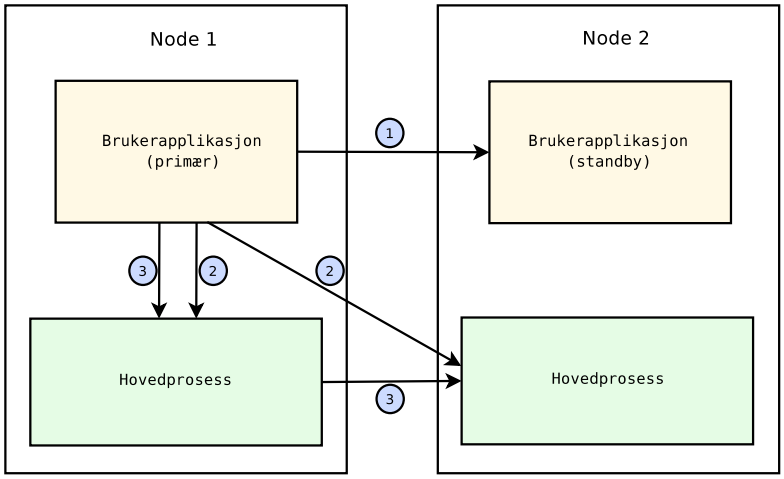
- Meldinger mellom brukerapplikasjon og hovedprosess
- Meldinger mellom hovedprosesser på ulike noder
- I'm-alive-meldinger mellom noder

Alle disse formene for kommunikasjon er helt essensielle for å oppnå ønsket funksjonalitet i løsningen. I implementeringen av [4] ble stream sockets benyttet for overføring av funksjonskall mellom brukerapplikasjon og hovedprosess. Denne teknologien baserer seg på TCP/IP protokollene (se kapittel 5.2). Fordelen med å velge denne teknologien var at man, i tillegg til å få en pålitelig to-veis forbindelse, kunne utnytte samme prinsipper for meldingssending mellom prosesser lokalt på en maskin som for meldingssending mellom prosesser på ulike noder. I den utvidede implementasjonen ble derfor stream sockets valgt også for dataoverføring mellom nodene.

Dette prosjektet gjør bruk av RPC (Remote Procedure Calls) som er en teknikk for å konstruere distribuerte applikasjoner. RPC er en utvidet form for lokale metodekall, og åpner for at metoden som kalles ikke trenger å ligge i samme adresserom som metoden den kalles fra. De to prosessene kan være i samme system, eller på ulike systemer knyttet sammen i et nettverk. Funksjonskallene til AIS som gjøres fra brukerapplikasjonen må sendes til hovedprosessen slik at denne kan prosessere forespørselen fra sin klient. Dette gjøres ved at en tallkode som identifiserer funksjonskallet sendes over socketforbindelsen sammen med innparametrene til funksjonskallet. Hovedprosessen leser funksjonsidentifikatoren og bestemmer ut fra denne hvilke operasjoner som skal foretas. Eventuelle innparametre leses også fra socketen og brukes videre i de aktuelle operasjonene. Dersom funksjonskallet til AIS har en utparameter eller returverdi, sendes denne motsatt vei over socketforbindelsen straks hovedprosessen er ferdig med prosesseringen av forespørselen.

Datautveksling mellom hovedprosesser på to noder dreier seg om checkpointdata fra primær til standby, samt bekreftelse fra standby til primær om at checkpointingen er utført. Ved valg av vei for checkpointdataene stod det mellom tre alternativer (se figur 6.2):

1. Checkpointdata sendes direkte mellom to brukerapplikasjoner og lagres i disse prosessenes adresserom.
2. Checkpointdata sendes fra brukerapplikasjon til begge hovedprosessene og lagres i hovedprosessenes adresserom.
3. Checkpointdata sendes fra brukerapplikasjon til hovedprosessen på samme node, som i sin tur sender disse videre til hovedprosessen på standbynoden. Dataene lagres i hovedprosessenes adresserom.



Figur 6.2: Alternativer for overføring av checkpointdata mellom nodene

Sending av data over flere ledd vil føre til redusert ytelse, og med hensyn til dette kriteriet peker alternativ 1 seg ut som den beste løsningen. Her opprettes det en egen socketforbindelse direkte mellom de to brukerapplikasjonene, og datastrukturene som representerer checkpointet lagres i brukerappliksjonens adresserom. Dette innebærer at selve checkpointingen foregår bak kulissene for hovedprosessen. I henhold til AIS standarden skal det imidlertid være mulig å la brukerapplikasjonen kjøre på kun én av nodene samtidig som checkpointing av data foregår. Alternativ 1 vil ikke tillate dette og ble derfor utelukket.

Alternativ 2 og 3 vil begge tillate checkpointing uavhenig av om et replikat av brukerapplikasjonen kjører på standbynoden. Mengden data som må sendes er den samme for begge alternativer, men alternativ 3 kan være utsatt for en viss latens siden dataene som sendes fra brukerapplikasjonen må leses en gang hos hovedprosessen på primærnoden før de sendes videre. Med alternativ 3 er det imidlertid enklere å sikre at checkpointdataene enten skrives i sin helhet hos begge hovedprosessene eller at de ikke skrives i det hele tatt. I tillegg vil

man ved bruk av dette alternativet unngå å måtte opprette en ekstra socketforbindelse mellom brukerapplikasjonen på primærnoden og hovedprosessen på standbynoden. Alternativ 2 vil altså medføre en ekstra socketforbindelse pr standbynode, noe som vil innebære mer arbeid for brukerapplikasjonen i forbindelse med takeover, særlig dersom man tenker seg flere noder i clusteret. Brukerapplikasjonen på den noden som går over til å bli primær, ville ved takeover først bli nødt til å kartlegge hvilke noder som inngår i clusteret, for så å opprette en forbindelse til hovedprosessene på hver av disse. Ved bruk av alternativ 3 vil man kunne utnytte en socketforbindelse som allerede eksisterer. Forbindelsen mellom hovedprosessene vil i alle tilfeller være nødvendig for annen kommunikasjon og synkronisering. Derfor ble det valgt å implementere dette alternativet.

For å kunne formidle I'm-alive-meldinger mellom de to nodene, benytter den implementerte løsingen seg som nevnt av cluster programvaren Heartbeat. Heartbeat kan blant annet konfigureres med hensyn på hvilken frekvens I'm-alive-meldingene skal sendes ut med, noe som vil være avgjørende for hvor raskt et nodekrasj kan detekteres.

6.3 CKPT funksjoner og relaterte datastrukturer

Løsningen som er implementert inkluderer et utvalg av AMF og CKPT funksjoner som er nødvendige for å realisere en feiltolerant brukerapplikasjon som innehar en tilstand. AMF biblioteket må initialiseres for å kunne benytte andre funksjoner i rammeverket. De nødvendige delene av dette er implementert i [4], og benyttet i dette prosjektet. Hvis en applikasjon skal kunne inneha ulike tilstander, vil det i en feilsituasjon forutsettes at checkpointing er implementert for at takeover skal kunne gjennomføres korrekt. Dersom en feilsituasjon oppstår, og standbynoden må overta for primærnoden, er det nødvendig å finne tilbake til den siste tilstanden som eksisterte før feilen ble gjeldende. På denne måten kan standbynoden fortsette utførelsen av programmet på en korrekt måte. Den benytter da de nyeste dataene som ble checkpointet av primærnoden før krasjet oppsto, da disse verdiene ikke er påvirket av feilen.

Aspektene som introduseres ovenfor representerer et sett med egenskaper løsningen må inneha for å tilfredsstille kravene oppgaven fordrer. Disse egenskapene impliserer realiseringen av CKPT funksjonene beskrevet i kapittel 6.3.1. Relaterte datastrukturer presenteres i kapittel 6.3.2. Sammenhengen mellom de ulike funksjonene i CKPT biblioteket er illustrert i figur 6.6.

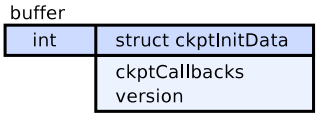
6.3.1 CKPT funksjoner

saCkptInitialize()

Funksjonen initialiserer *Checkpoint Service* tjenesten, samt registrerer ulike callbackfunksjoner, for den prosessen som sender kall til *saCkptInitialize()*. Brukeren av tjenesten må kalle *saCkptInitialize()* før den kaller noen andre funksjoner i *Checkpoint Service*. Hver initialisering returnerer en unik *handle* som benyttes til kommunikasjon med tjenesten. Dette er et heltall som fungerer som en

identifikator for den aktuelle prosessen. Slike identifikatorer brukes for å betegne en bestemt initialisering. Tildelingen av *handles* skjer ved hjelp av en såkalt *handlepool* (se kapittel 6.3.2).

Applikasjonsprosessen kommuniserer med *Checkpoint Service* via en socket-forbindelse til hovedprosessen. Når en applikasjonsprosess kaller *saCkptInitialize()*, vil det sendes en melding til tjenesten over denne forbindelsen. Meldingen identifiseres unikt ved et heltall, konstanten CKPTINITIALIZE, slik at riktig handling kan utføres hos hovedprosessen. Denne identifikatoren sendes over sammen med parameterlisten, som er pakket i en struct (se figur 6.3).



Figur 6.3: Pakking av data tilhørende *saCkptInitialize()*

I funksjonskallet angis hvilke tilgjengelige *callback*funksjoner som eksisterer for checkpointet. Parameteren *ckptCallbacks* er en struct som inneholder de callbackene applikasjonen kan benytte. Dersom denne parameteren er satt til NULL er det ingen registrerte callbacks. I dette prosjektet er det ikke implementert noen callbackfunksjoner i forbindelse med *Checkpoint Service*. Dersom asynkron skriving skal implementeres i en utvidelse av prosjektet, vil dette være nødvendig (se neste avsnitt om *saCkptCheckpointOpen()*). Parameteren *version* angir hvilken versjon av *Checkpoint Service* som benyttes. Denne variabelen er det ikke tatt hensyn til i dette prosjektet, da det kun er utviklet én tilgjengelig versjon av tjenesten.

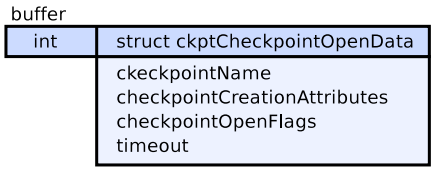
saCkptCheckpointOpen()

Denne funksjonen oppretter et nytt checkpoint, eller åpner et som eksisterer fra før. Kallet til *saCkptCheckpointOpen()* er blokkerende, det vil si at det ikke kan utføres andre kall før funksjonen har returnert med et svar. Dersom funksjonskallet gjennomføres feilfritt, vil det returnere en unik identifikator (*handle*) til checkpointet som er åpnet eller opprettet. Et checkpoint kan åpnes flere ganger for lesing og/eller skriving av ulike prosesser.

Et checkpoint kan åpnes eller opprettes både synkront og asynkront (se kapittel 4.5.1). I dette prosjektet er det valgt å implementere kun synkron opprettelse av checkpoints. Ved synkron oppdatering vil enten alle eller ingen checkpoints oppdateres med ny verdi. Asynkron oppdatering tillater derimot at kun det aktive checkpoint replikatet oppdateres. Deretter oppdateres de andre replikatene asynkront med den nye verdien. Fordelen med synkron oppdatering er at dataene på replikatene forblir konsistente, da alle eller ingen checkpoints oppdateres. Ulempen er latenstiden metoden medfører siden programmet må vente til dataene er skrevet på alle replikater. Dersom asynkron oppdatering ble benyttet, kunne man skrevet til det aktive checkpointet og deretter fortsatt utførelsen av programmet uten å oppdatere de resterende replikatene. Disse ville blitt oppdatert asynkront ved ledig kapasitet, og dermed gitt en raskere utførelsestid.

Ulempen med metoden er at konsistens ikke er sikret. Skulle en feilsituasjon oppstå, for eksempel ved et nodekrasj, risikerer man at det er ulike checkpoint-verdier hos de ulike replikatene.

Programmeringen av synkron oppdatering vil være mye enklere og mindre tidkrevende å utføre. I tillegg er det ingen behov for asynkron oppdatering i det enkle testprogrammet som utvikles i dette prosjektet. Ved en eventuell utvidelse av programmet kan dette eventuelt suppleres til løsningen uten at andre endringer i programkoden er nødvendig. Dersom man skulle opprettet checkpointet asynkront, ville dette skjedd ved kall til metoden *saCkptCheckpointOpenAsync()*. Denne returnerer med en gang, slik at applikasjonen kan fortsette programutførelsen uten å vente på ferdigstillelse av metoden. Når *saCkptCheckpointOpenAsync()* har utført det som er nødvendig for å opprette et asynkront checkpoint, kalles en callbackfunksjon så applikasjonen får beskjed om dette.



Figur 6.4: Pakking av data tilhørende *saCkptCheckpointOpen()*

Parameterlisten som sendes over i funksjonskallet er illustrert i figur 6.4. Parameteren *ckeckpointName* er navnet som identifiserer checkpointet globalt i clusteret. Dette er unikt for hvert enkelt checkpoint. *checkpointCreationAttributes* er de attributtene checkpointet tildeles ved opprettelse. Disse er kun relevante ved opprettelse av et nytt checkpoint, og settes til NULL dersom hensikten er å åpne et som allerede eksisterer. Attributtene som settes er:

- creationFlags

  Denne brukes for å synkronisere skriveoperasjoner til checkpoint replikatene, og til å opprette og slette *sections* i et checkpoint. Attributtet kan inneha tre ulike verdier (se kapittel 4.5.1). I dette prosjektet er den satt til SA_CKPT_WR_ALL_REPLICAS. Grunnen til dette er at all skriving av checkpoints utføres synkront, og denne verdien er tilpasset dette formålet. Modifiseringer som skriving, opprettelse, sletting og overskrivning av checkpoints vil utføres på alle replikater før operasjonen returnerer med et resultat. Det garanteres også for atomitet for hvert replikat. Dersom det oppstår en feil under utførelsen, vil da ingenting bli skrevet. De andre to verdiene attributtet kan inneha er tilpasset asynkron skriving.

- checkpointSize

  Attributtet angir størrelsen som settes av til hvert checkpointreplikat i bytes. Da testapplikasjonen i dette prosjektet skriver ut heltall, er det naturlig å sette replikatstørrelsen til 4 bytes som er størrelsen på en *integer* (heltall).

- retentionDuration

  retentionDuration angir hvor lenge et checkpoint skal eksistere i systemet dersom det ikke har vært åpnet av en prosess i løpet av en gitt tidsperiode. Eksempelapplikasjonen som er utviklet i dette prosjektet skriver ut et nytt heltall hvert sekund som er inkrementert med 1 i forhold til foregående tall. For hver inkrementering checkpointes den nye verdien slik at tilstanden til applikasjonen lagres. Applikasjonen har checkpointet åpent så lenge denne prosessen pågår, det vil si hele dens levetid, og har i utgangspunktet ikke behov for dette attributtet. *retentionDuration* er satt til 20 sekunder, men det vil aldri oppstå en situasjon hvor denne verdien overstiges. Attributtet vil først være nyttig når det eksisterer flere checkpoints som kan åpnes og lukkes av ulike applikasjoner. Det vil da kunne oppstå opphopning av checkpoints dersom de ikke fjernes etter en gitt tidsperiode. I dette prosjektet opprettes det kun ett checkpoint som hele tiden overskrives med de nye verdiene. Det er derfor ingen risiko for at ubrukte checkpoints skal bli liggende i systemet.

- maxSections

  maxSections angir maksimalt antall *sections* som kan eksistere i et checkpoint. Denne verdien må være minst 1, da alle checkpoints har minst en *section*. Dersom verdien er 1, vil det automatisk opprettes en *default section* som kjennetegnes av identifikatoren SA_CKPT_DEFAULT_SECTION_ID. Sistnevnte løsning er valgt for testapplikasjonen. Slik applikasjonen foreligger vil det ikke være behov for flere *sections* innad i et checkpoint. *maxSections* er derfor satt til 1.

- maxSectionSize

  Attributtet angir øvre grense på størrelsen til en *section* i dette checkpointet. Da applikasjonen checkpointer heltall vil det ikke være behov for større *sections* enn størrelsen på en *integer*, det vil si 4 bytes.

- maxSectionIdSize

  maxSectionIdSize angir den maksimale lengden på identifikatoren til en *section*. Disse er angitt ved heltall, og derfor settes denne verdien til størrelsen på en *integer* (4 bytes).

Parameteren *checkpointOpenFlags* kan ta tre ulike verdier:

- SA_CKPT_CHECKPOINT_READ

  Angir at checkpointet åpnes i lesemodus

- SA_CKPT_CHECKPOINT_WRITE

  Angir at checkpointet åpnes i skrivemodus

- SA_CKPT_CHECKPOINT_COLOCATED

  Dersom det ikke eksisterer et checkpoint replikat på den lokale noden, og dersom *Checkpoint Service* ikke kan opprette et replikat her, returneres en feilmelding. Ellers oppretter tjenesten et lokalt checkpoint replikat på noden.

I dette prosjektet er verdien SA_CKPT_CHECKPOINT_WRITE benyttet ved opprettelse av et checkpoint. Grunnen til dette er at verdien som checkpointes stadig skal overskrives, og det er derfor nødvendig at det åpnes i skrivemodus.

timeout parameteren i kallet angir når man skal anta at kallet til funksjonen har feilet. Dersom funksjonen ikke har returnert et svar innen tidsangivelsen i *timeout* har kallet feilet og må eventuelt utføres på nytt.

saCkptCheckpointClose()

saCkptCheckpointClose() frigjør alle ressursene allokert av forrige kall til *saCkptCheckpointOpen()*. Funksjonen tar inn en *checkpointHandle*, som er en *handle* til det checkpointet som skal lukkes. Når kall til denne funksjonen er utført, vil bruk av handlen ikke lenger være gyldig. Dersom ingen andre prosesser åpner checkpointet etter dette, vil det bli slettet når oppbevaringstiden utløper (se kapittel 4.5.1). Når en prosess avsluttes, lukkes alle dens åpne checkpoints.

saCkptCheckpointWrite()

Denne funksjonen skriver data fra spesifiserte minneområder inn i et checkpoint. Minneområdene er angitt i en vektor kalt *ioVector*, en datastruktur som også brukes i forbindelse med *saCkptCheckpointRead()*, se neste avsnitt. Hvert element i *ioVector* omfatter følgende data og er illustrert i figur 6.5:

- sectionId

  Dette er en identifikator til den *sectionen* som skal skrives (eller leses). *sectionId* settes i dette prosjektet til SA_CKPT_DEFAULT_SECTION_ID som beskrevet over, da det kun benyttes én *section* per checkpoint.

- dataBuffer

  Variabelen inneholder en peker til de dataene som skal skrives (eller leses).

- dataSize

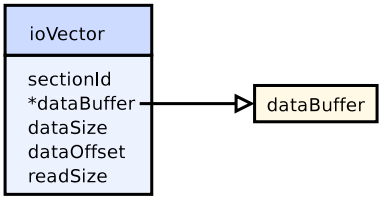
  dataSize angir størrelsen på dataene som skal skrives til (eller leses fra) *dataBuffer*. Maksimal størrelse dette bufferet kan ha angis i størrelsen *maxSectionSize* ved opprettelsen av et checkpoint (se *saCkptCheckpointOpen()*).

- dataOffset

  Markerer starten på dataene som skal skrives (eller leses) relativt til starten på seksjonen.

- readSize

  Denne brukes av metoden *saCkptCheckpointRead()* for å finne hvor mange bytes med data som er lest. I andre sammenhenger benyttes ikke dette feltet.



Figur 6.5: Illustrasjon av elementene i *ioVector*

Når et checkpoint opprettes ved *saCkptCheckpointOpen()*, settes et *creationFlag* som beskrevet over. Dette er en parameter som angir noen av egenskapene checkpointet innehar. Avhengig av hvilket parametervalg som er foretatt, vil skriving til et checkpoint utarte seg som følger:

- SA_CKPT_WR_ALL_REPLICAS

  Dersom denne egenskapen er satt, vil alle eksisterende checkpoint replikater være oppdatert når funksjonen returnerer. Dersom skrivingen ikke blir gjennomført eller funksjonen returnerer en feilmelding, vil ingen verdier bli skrevet til noen checkpoint replikater.

- SA_CKPT_WR_ACTIVE_REPLICA

  Opprettes checkpointet med denne parameteren, vil det aktive checkpoint replikatet være oppdatert når funksjonen returnerer. Andre checkpoint replikater oppdateres asynkront som beskrevet i kapittel 4.5.1. Dersom skrivingen ikke blir gjennomført eller funksjonen returnerer en feilmelding, vil ingen verdier bli skrevet til noen checkpoint replikater.

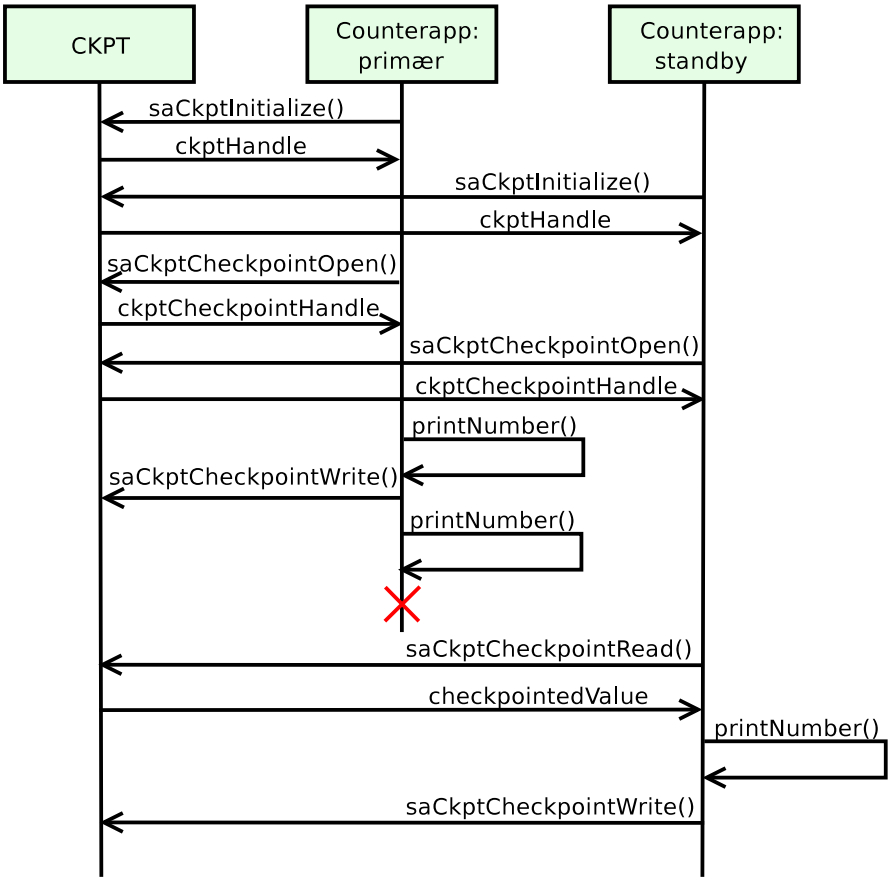
- SA_CKPT_WR_ACTIVE_REPLICA_WEAK

  Dersom checkpointet blir opprettet med denne egenskapen, vil det aktive checkpointet være oppdatert når funksjonen returnerer. Andre checkpoint replikater oppdateres asynkront. Dersom metoden returnerer en feilmelding, vil ingen verdier bli skrevet til noen checkpoint replikater. Hvis funksjonen derimot ikke blir ferdig med utførelsen kan operasjonen bli delvis gjennomført, og enkelte sections kan bli korrumpert i det aktive checkpoint replikatet.

Funksjonen har en utparameter kalt *erroneousVectorIndex*. Dette er en peker til en indeks, lagret i adresserommet til prosessen som kaller funksjonen, og inneholder det første *ioVector* elementet som gjør at kallet feiler. Dersom indeksen er satt til NULL, eller kallet lykkes, vil feltet forbli uendret.

saCkptCheckpointRead()

Denne funksjonen fungerer på tilsvarende måte som *saCkptCheckpointWrite()*. Den tar inn en *checkpointHandle* som angir hvilket checkpoint det skal leses fra. Deretter kopieres data fra checkpoint replikatet inn i vektoren spesifisert ved *ioVector*. Verdiene kan da leses av applikasjonen. På samme måte som *saCkptCheckpointWrite()* har funksjonen en utparameter kalt *erroneousVectorIndex* (se over).

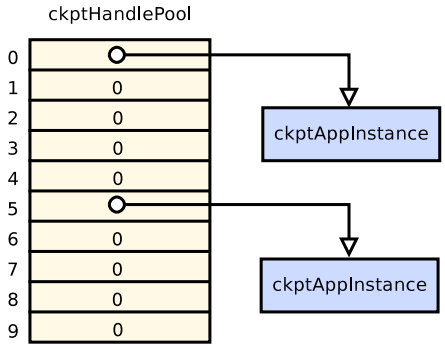


Figur 6.6: Bruk av funksjonene i CKPT biblioteket

6.3.2 Datastrukturer

ckptHandlePool

Funksjonen *saCkptInitialize()* returnerer en unik *handle* for hver initiering av biblioteket. For å administrere tildelingen av slike *handles*, er det opprettet et såkalt *handlepool*. Denne datastrukturen inneholder unike *handles* som kan tildeles applikasjonene som ønsker å benytte seg av tjenestene til *Checkpoint Service*. *ckptHandlePool* er realisert som en tabell hvor indeksen i tabellen representerer hver enkelt *handle*. Dataene som er lagret i hver post inneholder en peker til en *ckptAppInstance* (se neste avsnitt). Datastrukturen *ckptAppInstance* representerer hver applikasjon som registrerer seg hos *Checkpoint Service*. På denne måten kan *Checkpoint Service* holde rede på hvilke applikasjonsprosesser den er tilknyttet, samt innhente informasjon om deres egenskaper. En post i *handlepoolen* med verdien "0" tilsier at den *handle* som tilsvarer postens indeks er ledig for nye applikasjoner som ønsker å registrere seg. I figur 6.7 er *ckptHandlePool* illustrert. Her er to av indeksene i bruk, og *ckptAppInstance*-datastrukturene har fått henholdsvis handle 0 og 5.



Figur 6.7: *ckptHandlePool* inneholder pekere til en *ckptAppInstance* for hver *handle* som er i bruk.

Ved å benytte en slik datastruktur begrenses antall feiltolerante applikasjoner som kan ha kontakt med *Checkpoint Service* til samme tid. Grunnen til at det er utført på denne måten er at AMF, som bestemmer antall applikasjoner som kan eksistere totalt sett, har en begrensning på hvor mange brukerapplikasjoner den kan håndtere samtidig (se [4]).

ckptAppInstance

Når funksjonen *saCkptInitialize()* kalles, opprettes en datastruktur kalt *ckptAppInstance* (se figur 6.8) hos hovedprosessen. En instans av denne datastrukturen representerer hver applikasjon som registrerer seg hos *Checkpoint Service*, og inneholder informasjon om følgende:

- ckptHandle

  Denne variabelen er et heltall som tildeles en prosess under initialisering av *Checkpoint Service*. Verdien er tildelt av *ckptHandlePool* og brukes av prosessen når den kaller funksjoner i *Checkpoint Service* APIet, slik at tjenesten kan identifisere prosessen.

- ckptCallbacks

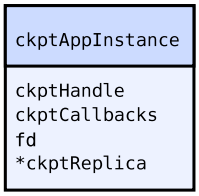
  Her ligger en oversikt over hvilke callbacks som implementeres av applikasjonen

- fd

  fd er en fildeskriptor som identifiserer socketforbindelsen til applikasjonen. Mer informasjon om socketkommunikasjon finnes i kapittel 5.2.

- *ckptReplica

  Dette er en peker til instansen *ckptCheckpointReplica* (se under). Det opprettes et checkpoint replikat for hver applikasjon som ønsker å benytte seg av *Checkpoint Service*.

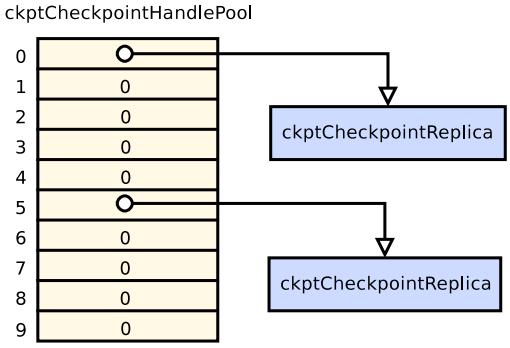


Figur 6.8: Data tilhørende *struct ckptAppInstance*

ckptCheckpointHandlePool

Funksjonen *saCkptCheckpointOpen()* returnerer en unik *handle* for hvert nytt checkpoint replikat som opprettes. Det er opprettet et *handlepool* for å administrere tildelingen av slike *handles* (se figur 6.9). Denne datastrukturen er realisert på samme måte som handlepoolen beskrevet over. Dataene som er lagret i hver post inneholder en peker til et *ckptCheckpointReplica* (se neste avsnitt). Datastrukturen *ckptCheckpointReplica* representerer hvert checkpoint som opprettes av applikasjonene. På denne måten kan *Checkpoint Service* holde rede på hvilke checkpoint som er tilknyttet de ulike applikasjonsprosessene, samt innhente informasjon om prosessenes egenskaper.

For testapplikasjonen som er implementert i dette prosjektet er det kun aktuelt med ett checkpoint per applikasjon. Det vil derfor ikke være nødvendig med flere indekser i *handlePoolen* enn antall applikasjoner som kan opprettes. Da begrensningen er satt til 10 applikasjoner, har også *ckptCheckpointHandlePool* 10 indekser. I en eventuell utvidelse av programmet vil disse verdiene enkelt kunne endres i konfigurasjonsfilen.



Figur 6.9: *ckptCheckpointHandlePool* inneholder pekere til et *ckptCheckpoint-Replica* for hver *handle* som er i bruk.

ckptCheckpointReplica

Ved kall til funksjonen *saCkptCheckpointOpen()* kan man velge om det skal opprettes et nytt checkpoint, eller om et eksisterende checkpoint skal åpnes.

Dersom et nytt checkpoint skal lages, opprettes det et checkpoint replikat representert ved datastrukturen *ckptCheckpointReplica* (se figur 6.10). Denne har følgende attributter:

- ckptCheckpointHandle

  Et heltall som unikt representerer et checkpoint replikat.

- **sections

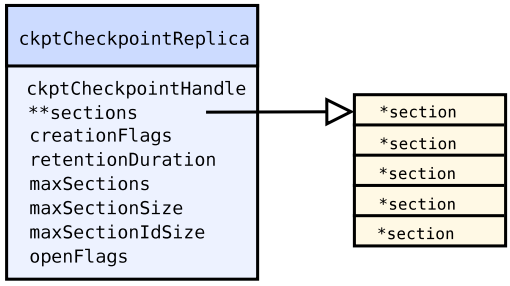
  Hvert checkpoint består av en eller flere *sections*. Det er derfor opprettet en pekertabell hvor hvert av elementene peker til en slik *ckptSection* (se neste avsnitt). Det må minst eksistere en *section* i hvert checkpoint.

- creationFlags, retentionDuration, maxSections, maxSectionSize og maxSectionIdSize

  Disse parametrene er en del av *ckptCreationAttributes* som settes ved opprettelse av et checkpoint. En nærmere gjennomgang av disse finnes under beskrivelsen av funksjonen *saCkptCheckpointOpen()*.

- openFlags

  openFlags angir hvilket modus checkpointet skal åpnes i. De ulike verdiene den kan inneha er oppført under beskrivelsen av funksjonen *saCkptCheckpointOpen()*.



Figur 6.10: Data tilhørende *struct ckptCheckpointReplica*

ckptSection

Hver *section* innad i et checkpoint har et sett med egenskaper som lagres i en *struct* (se figur 6.11). Datastrukturen har følgende attributter:

- sectionId

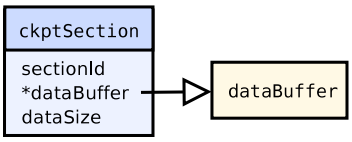
  Hver section har et heltall, *sectionId*, som unikt identifiserer det innenfor et checkpoint. *sectionId* settes i dette prosjektet til SA_CKPT_DEFAULT_SECTION_ID som beskrevet over, da det kun benyttes en *section* per checkpoint.

- *databuffer

  Dette er en peker til minneområdet hvor dataene tilhørende denne *sectionen* befinner seg.

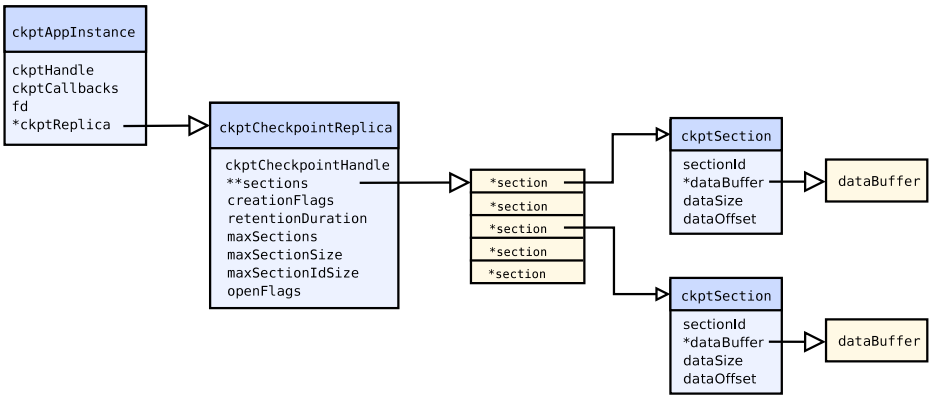
- dataSize

  Variabelen *dataSize* angir størrelsen på *dataBuffer*. Størrelsen kan maksimalt være *maxSectionSize*, som angis ved opprettelse av et checkpoint.



Figur 6.11: Data tilhørende s*truct ckptSection*

Alle de ovenfor nevnte datastrukturene avhenger av hverandre, og er nødvendige for realiseringen av *Checkpoint Service*. Sammenhengen mellom disse er illustrert i figur 6.12.



Figur 6.12: Sammenhengen mellom alle datastrukturene brukt i realiseringen av *Checkpoint Service*

6.4 Detaljert beskrivelse av løsningen

Ved oppstart av programmet oppretter hovedprosessen en primær og en standby-versjon av brukerapplikasjonen på to uavhengige noder. Det er tilrettelagt for at inntil fem applikasjoner kan kjøre i systemet samtidig. I tilknytning til applikasjonsfilene er det opprettet en konfigurasjonsfil. Her legges stiene til hver av brukerapplikasjonene som skal benytte seg av AIS. Filene leses av hovedprosessen og denne starter opp en primær og en standby per sti slik at man oppnår to kjørende versjoner av hver applikasjon. Disse vil befinne seg på to uavhengige maskiner, noe som muliggjør høytilgjengelighet.

Den totale programflyten i den implementerte løsningen involverer kommunikasjon med AMF og CKPT, og er i sin helhet illustrert i figur 6.16. Beskrivelsen av kommunikasjonen med AMF er gitt i [4], og vil ikke bli detaljert belyst i dette avsnittet. Her vektlegges kommunikasjonen med CKPT da dette, i tillegg til integrasjon av Heartbeat med AIS, er prosjektets fokusområde.

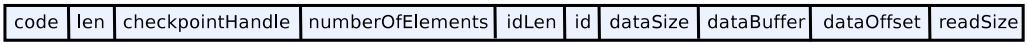
Begge applikasjonene initialiserer alle tilgjengelige deler av biblioteket ved oppstart. Dette skjer først ved bibliotekskallet *saAmfInitialize()*. I denne funksjonen opprettes socketforbindelsen mellom den aktuelle applikasjonsprosessen og dens tilhørende hovedprosess. Dette er en forutsetning for at kommunikasjon mellom de to prosessene skal finne sted. Fra *saAmfInitialize()* returneres en *handle* som er generert av hovedprosessen til applikasjonsprosessen. Denne benytter hovedprosessen for å skille mellom de ulike applikasjonsprosessene den kommuniserer med. Deretter utføres bibliotekskallet *saCkptInitlialize()*. Her benyttes den samme kommunikasjonskanalen som ble opprettet ved initialiseringen av AMF. Dette er mulig å gjøre da kallene til *Checkpoint Service* kun skjer ved oppstarten av programmet, og dermed ikke skaper konflikt med kall til AMF. Dersom det i en utvidelse av prosjektet blir aktuelt å implementere callbacks og *saCkptDispatch()*, kan dette løses ved å benytte to ulike socketforbindelser. *Dispatch*-funksjonene forsøker å lese fra socketen for å se om nye data ligger på vent. Eventuelle data må deretter pakkes ut og tolkes. Når et callback har kommet inn, vil applikasjonsprosessen finne fram til riktig funksjonspeker, for så å kalle denne funksjonen under kjøring. Dersom det ikke implementeres to socketforbindelser, vil det være tilfeldig om meldingen leses av *saAmfDispatch()* eller *saCkptDispatch()*. Disse skal utføre kall til ulike callbackfunksjoner, og det er derfor nødvendig at meldingen havner hos riktig *dispatch*-funksjon. En alternativ løsning vil være å sende en identifikator med dataene som sier om de tilhører *saAmfDispatch()* eller *saCkptDispatch()*. Når meldingen ankommer hovedprosessen legges den da i en kø tilsvarende funksjonen den skal kalle. *Dispatch*-funksjonene leser da fra sine respektive køer og utfører metodekallene.

Det neste som skjer er at hver av applikasjonsprosessene kaller *saCkptCheckpointOpen()* for å opprette et nytt checkpoint (se kapittel 6.3.1). Det returneres da en *handle* kalt *ckptCheckpointHandle* fra hovedprosessen. Denne brukes for å skille de ulike checkpointene fra hverandre. Hver av applikasjonsprosessene registrerer videre en komponent ved hjelp av *saAmfComponentRegister()*. Begge komponentene har i utgangspunktet *readiness state* satt til *out-of-service* (se [4]). Når den første applikasjonsprosessen har registrert sin komponent hos hovedprosessen, sender denne et callback tilbake med beskjed om å sette *readiness state* til *in-service*. Denne blir da primær, og kaller *saCkptCheckpointRead()* som returnerer siste checkpointede tilstand. Når programmet startes opp første gang, og ingen prosesser tidligere har utført tellerapplikasjonen, returnerer *erroneousVectorIndex* -1. *erroneousVectorIndex* brukes også for å ta høyde for at primærprosessen kan krasje før den rekker å starte opp tellerapplikasjonen, og dermed ikke får checkpointet noen verdi. Dersom primærprosessen aldri får skrevet ut første tall før den går ned, og dermed ikke får utført checkpointing av en verdi, må standbyprosessen allikevel vite hvor den skal starte opp ved *takeover*. Dersom det ved oppstart leses en tom sekvens istedet for en checkpointverdi tolkes dette som tallet 0. Dermed vet applikasjonen hvor den skal starte programutførelsen fra. Applikasjonsprosessene med *readiness state* satt til *out-of-service* betraktes som standbys. I denne tilstanden ligger de og venter

på at et callback skal komme fra hovedprosessen, noe som realiseres ved kall til *saAmfDispatch()* en gang i sekundet. I det en komponents *readiness state* endres til *in-service* blir dens tilstand satt til primær, og den starter å skrive ut itererende tall fra det punktet forrige prosess slapp. Denne situasjonen oppstår dersom den opprinnelige primærprosessen krasjer og dermed slutter å utføre sine oppgaver. Det nye startpunktet finnes ved kall til funksjonen *saCkptCheckpointRead()*. Denne returnerer siste checkpointede tilstand fra prosessen som krasjet. Checkpointverdien ligger lagret i hovedprosessens adresserom. Dataene som sendes over fra klientapplikasjonen til hovedprosessen ved lesing av et checkpoint er illustrert i figur 6.13. Hovedprosessen returnerer da tilbake den siste checkpointede verdien som er lagret i dens adresserom.

Figur 6.13: Oversending av data fra *saCkptCheckpointRead()* til server

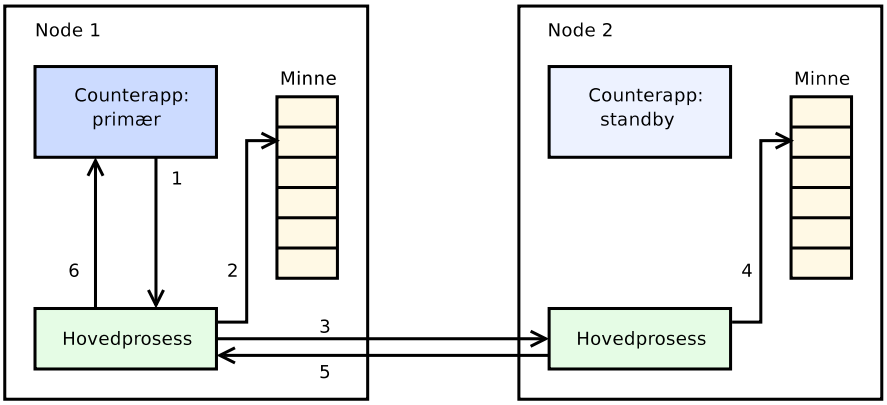
Når primærprosessen har skrevet ut et tall til skjermen, checkpointes denne verdien hos hovedprosessen ved kall til *saCkptCheckpointWrite()*. Dataene som sendes over til hovedprosessen ved skriving av et checkpoint er illustrert i figur 6.14.



Figur 6.14: Oversending av data fra *saCkptCheckpointWrite()* til server

Disse dataene sendes først til lokal hovedprosess og deretter videre til hovedprosessen på standbynoden. Dette er fordi både primær- og standbynoden må ha tilgang på checkpoint dataene for å kunne innhente informasjon om hvilken tilstand applikasjonen befinner seg i til enhver tid. Hvordan skriving av checkpoint verdier til primær- og standbynode utføres, er illustrert i figur 6.15.

Først sendes checkpoint dataene fra primærprosessen til dens hovedprosess. Dataene sendes over socketforbindelsen som ble opprettet i *saAmfInitialize()*. Disse tas så imot hos hovedprosessen og lagres i dens adresserom ved hjelp av datastrukturene beskrevet i kapittel 6.3.2. På denne måten har hovedprosessen alltid tilgang til checkpoint dataene, og kan formidle disse tilbake til klientapplikasjonen ved utførsel av leseoperasjoner. Etter at den lokale skrivingen er foretatt, sendes checkpointverdiene over til standbyapplikasjonens hovedprosess via en socketforbindelse. Socketforbindelsen ble opprettet i oppstarten av programmet, og benyttes til oversendelse av data ved checkpointing av en ny verdi. Når meldingen er mottatt av standbynoden pakkes denne ut, og lagres i adresserommet til standbynodens hovedprosess. Deretter sendes et svar tilbake om at checkpointingen er gjennomført. Når hovedprosessen på primærnoden mottar denne responsen sender den beskjed til primærprosessen om at checkpointingen var vellykket. Siden det er benyttet synkron skriving i dette prosjektet, skal dataene enten skrives til alle noder, eller ikke skrives i det hele tatt. Det er derfor nødvendig med tilbakemelding angående vellykket sending og korrekt mottakelse av data.



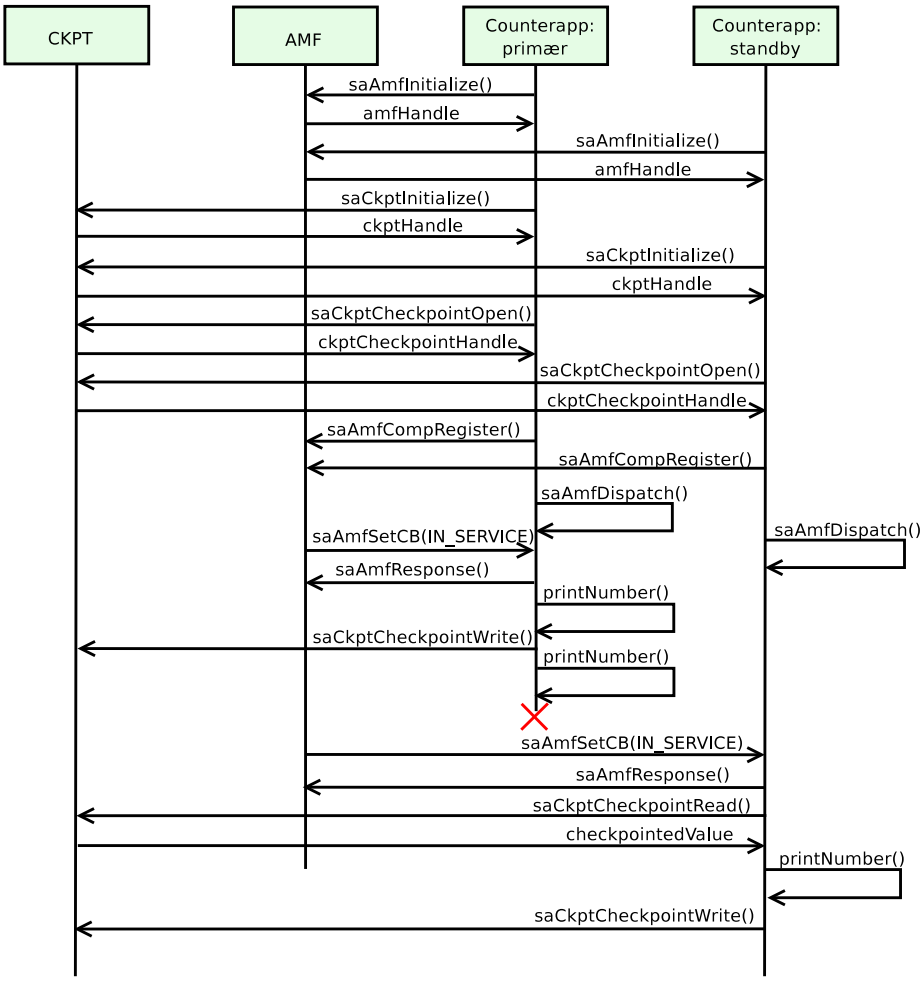
Figur 6.15: Skriving av checkpointdata til primær- og standbynode

På server-siden av socketforbindelsen er det ønskelig med ikke-blokkerende kommunikasjon. Da unngår man at hovedprosessen blir stående og vente på at en spesifikk hendelse skal inntreffe, og dermed blokkere annen programutførelse. Funksjonskallet *select()* er benyttet i realiseringen av dette på serversiden. *select()* vil detektere hvilke fildeskriptorer det har skjedd en endring på ved det tidspunktet den kalles. For fildeskriptorer som har ventende data, leses disse og tolkes av hovedprosessen. Dataene forteller hvilken funksjon med tilhørende parameterverdier som skal kalles hos hovedprosessen. På denne måten kan applikasjonsprosessen sende meldinger til hovedprosessen, og oppnå at korrekte handlinger blir utført i henhold til den aktuelle forespørselen.

Socketforbindelsen mellom de to hovedprosessene er også lagt til i *select()*-funksjonen. Dermed detekteres også hendelser som har forekommet på denne kommunikasjonskanalen ved at tilhørende fildeskriptor identifiseres. Hovedprosessene har behov for å sende meldinger til hverandre i forbindelse med utførelsen av checkpointing som er beskrevet ovenfor. Data oversendes, leses og tolkes, før riktig funksjon med angitte parameterverdier utføres på motsvarende hovedprosess.

Ved tre tilfeller er det benyttet blokkerende kommunikasjon i utførelsen av prosjektet. Det ene tilfellet er ved applikasjonens kall til *saCkptCheckpointWrite()*. Når checkpointdata sendes over til standbynoden for skriving, skal det returneres SA_OK dersom skrivingen var vellykket. Da dette prosjektet anvender synkron skriving, er det nødvendig å innhente denne informasjonen før utførelsen av programmet kan holde frem. Av den grunn settes kommunikasjonen til å være blokkerende når det ventes på svar fra standbynoden angående utførelsen av checkpoint skriving. Det andre tilfellet hvor blokkerende kommunikasjon blir benyttet er ved kall til samme funksjon, men hvor skrivingen foregår lokalt. Applikasjonsprosessen venter da på svar fra serveren om at checkpointingen er gjennomført, før den fortsetter å utføre sine oppgaver. Årsaken til dette er at applikasjonsprosessen da kan være sikker på at checkpointingen er utført på alle nodene i clusteret. Som illustrert i figur 6.15 vil ikke hovedprosessen på primærnoden returnere svar før skriving av checkpoint er utført både lokalt og på standbynoden. Applikasjonsprosessen kan dermed være sikker på at synkron

skriving er gjennomført. Det siste tilfellet hvor blokkerende kommunikasjon er benyttet er ved kall til *saCkptCheckpointRead()*. Her er det naturlig å ha en slik funksjonalitet. Dersom primærnoden går ned og standbynoden tar over, er sistnevnte nødt til å lese siste checkpointede verdi før den starter opp. Da alle andre oppgaver som skal utføres avhenger av nettopp dette, må lesingen gjennomføres før noe annet kan skje. Den er derfor satt til å være blokkerende slik at videre utførelse av programmet forhindres før denne verdien er hentet.



Figur 6.16: Hendelsesforløp i programmet

6.5 Feilklasser

Begrepet feilklasser og tilhørende teori ble introdusert i avsnitt 2.3.1. Det gis her en kort oversikt over hvordan disse begrepene kan tolkes i forbindelse med implementasjonen utført i dette prosjektet.

Standardsemantikken til systemet tilsier at primærnoden skal utføre oppgaver kontinuerlig, mens standbynoden står parat til å ta over dersom førstnevnte feiler. I feilsemantikken er følgende klasser av feil tatt høyde for:

- Utelatelsesfeil

  Dette er feil som oppstår når serveren ikke gir respons på forespørsler, for eksempel at kommunikasjonstjenesten mister enkelte meldinger. I dette prosjektet er TCP/IP benyttet til kommunikasjon mellom nodene og mellom applikasjon og hovedprosess. Dersom dataene under overføring blir skadet, tapt eller duplisert, eller at rekkefølgen blir endret, håndteres dette av TCP/IP. Meldinger som mistes under overføring mellom noder maskeres av kommunikasjonstjenesten, og på denne måten er utelatelsesfeil tolerert i systemet.

- Krasj

  Krasjfeil oppstår når primærnoden slutter å utføre sine tjenester, for eksempel når strømforbindelsen til maskinen brytes. Dette håndteres ved at standbynoden får beskjed av Heartbeat om å ta over, og fortsetter utførselen fra der primærnoden sluttet. Dersom det skulle oppstå en situasjon som medfører at begge nodene krasjer, for eksempel et totalt strømbrudd eller en nettverksfeil, vil ikke dette bli håndtert av systemet. Det er ikke tatt høyde for at slike feil skal oppstå, og dette er dermed utenfor servernes feilsemantikk.

Det er ikke tatt hensyn til timingfeil eller responsfeil. Skulle allikevel slike feil oppstå, vil situasjonen være utenfor feilsemantikken definert for systemet. Det er vanskelig å estimere noen sannsynlighet for hvorvidt dette vil oppstå, men den anslås til å være relativt liten.

Kapittel 7

Beskrivelse av testene

I dette kapitlet beskrives testene som er utført for å måle tids- og ressursforbruket til checkpointingen. Det er to tilfeller som er interessante å studere med tanke på tidsforbruket til applikasjonen. Disse er feilfri kjøring, hvor alle operasjoner utføres som normalt, og kjøring hvor det oppstår en feil slik at primærnoden krasjer. I sistnevnte tilfelle vil det foregå en takeover, og tiden dette tar vil indikere om løsningen kan sies å være høytilgjengelig. Utførelsen av testene ved de to ulike tilfellene er beskrevet i påfølgende delkapitler.

7.1 Feilfri kjøring

Feilfri kjøring vil si at systemet oppfører seg som forventet. Det oppstår ingen uventede feil, og det er kun primærnoden som utfører oppgaver. I en slik situasjon er det ikke behov for takeover, men standby vil allikevel stå i posisjon til å ta over for primærnoden. Standby initialiserer alle sine biblioteker, og checkpointet skrives på begge noder. Dette er fordi tidtakingen skal bli helt realistisk og kunne sammenliknes med verdiene for kjøring med failover.

For å utføre testene benyttes det buffer av ulike størrelser, fra 1 til 10 000 bytes, som allokeres under kjøring av programmet. Dataene i bufferet representerer en tenkt tilstand til en applikasjon, og sendes over til standbynode i forbindelse med checkpointing. 10 000 bytes vil normalt være nok for å simulere tilstanden til en enkel applikasjon. Applikasjoner som ville kreve checkpoints utover denne størrelsen, bør benytte andre teknikker for å ta vare på tilstanden for å unngå for mye overhead ved feilfri kjøring. Pseudokoden som angir utførelsen av testene er vist i figur 7.1.

Ved bruk av systemkallet *gettimeofday()* returneres et tidspunkt, med nøyaktighet i mikrosekunder, som gjenspeiler tid på dagen. Dette kallet utføres før og etter kallet til *saCkptCheckpointWrite()* i klientapplikasjonen, og deretter finnes differansen mellom disse tidene. På denne måten finner man hvor stort tidsforbruk skrivingen av checkpointet medførte. På tilsvarende måte måles CPU-forbruket før og etter kallet til *saCkptCheckpointWrite()* ved kall til funksjonen *getrusage()*. Denne returnerer mengden CPU-ressurser prosessen har brukt i

form av to parametre; *systemtime* og *usertime*. *systemtime* sier noe om hvor mye tid som brukes på CPU-syklene i operativsystemkoden. For å eventuelt redusere dette tidsforbruket må antall systemkall reduseres. *usertime* gir informasjon om hvor mye tid som er brukt i applikasjonskoden, for eksempel i interne funksjonskall og løkker. Ved å regne ut differansen mellom disse verdiene før og etter checkpointingen, finnes det totale CPU-forbruket som gikk med under skriving av hvert checkpoint.

For å simulere en virkelig applikasjon må en viss mengde CPU-tid forbrukes under kjøring av testapplikasjonen. Dette er realisert ved å ha en løkke som går i 1 000 iterasjoner og summerer opp tallene fra 1 til 1 000. Denne delen av programmet har ingen funksjonalitet foruten simuleringseffekten, og vil derfor ikke inngå i målingene av tid og CPU-forbruket.

```
for(buffer = 1, 10, 100, 1000, 10000 bytes) {
    <alloker plass til bufferet>

    for(i = 1000 repetisjoner) {
        start_cpu = getrusage()
        start_tid = gettimeofday()

        saCkptCheckpointWrite()

        slutt_cpu = getrusage()
        slutt_tid = gettimeofday()

        <finn differansen mellom start- og slutt_tid>
        <finn differansen mellom start- og slutt_cpu>
        <skriv verdiene til fil>

        for(j = 1000 repetisjoner) {
            <summer 1000 tall>
        }
    }

    <frigjør den allokerte bufferplassen>
}
```

Figur 7.1: Pseudokode for måling av tid og CPU forbruk

Det utføres 1 000 målinger av tids- og CPU-verdier for hver størrelse av bufferet. Det er nødvendig med en stor mengde målinger for å kunne utføre statistiske analyser og trekke fornuftige konklusjoner på grunnlag av disse. Resultatet fra kjøringene skrives til en fil, hvor de senere kan hentes frem for videre prosessering.

7.2 Kjøring med failover

Etter å ha kartlagt hvor mye overhead selve checkpointingen ved feilfri kjøring vil medføre, er det interessant å anslå hvor mye tid det faktisk vil gå fra den ene noden feiler til den andre noden har detektert feilen og oppdatert sin tilstand

slik at den er klar til å kjøre fra det punktet der den første slapp. Det er ønskelig å undersøke tre ulike scenarier som kan forårsake takeover:

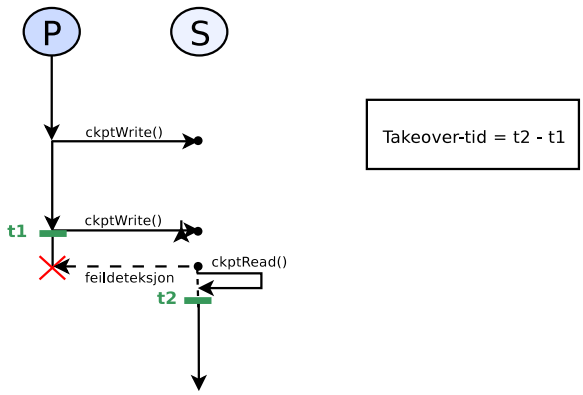
1. Applikasjonen feiler: krasjet detekteres av hovedprosessen på den aktuelle noden.
2. Hovedprosessen feiler: krasjet detekteres av hovedprosessen på standby-noden.
3. Nodekrasj: Heartbeat detekterer at en node i clusteret har gått ned.

For alle disse tilfellene ønsker man å måle takeover-tid. For alternativ 3 er det opplagt at takeover-tiden vil avhenge sterkt av den tiden Heartbeat bruker på å detektere at noden er nede. Derfor er det interessant å tune på Heartbeat-parametrene for å oppnå minimal deteksjonstid.

Takeover-tid baserer seg på differansen mellom to tidspunkt t1 og t2 (se figur 7.2), hvor

t1 = tid registrert etter at siste skriving av checkpoint hos primærnoden er gjennomført

t2 = tid registrert etter at standby har lest inn siste checkpoint og står klar for prosessering fra korrekt tilstand



Figur 7.2: Takeover-tid

For å kunne beregne takeover-tiden er det nødvendig å registrere tidspunktet i etterkant av hver checkpoint-skriving på primærnoden. I tillegg må tidspunktet registreres etter at applikasjonen på standbynoden har lest inn checkpointdata i forbindelse med takeover. De to tidspunktene som er interessante kommer følgelig fra to ulike noder og for å kunne beregne riktig takover-tid er man dermed avhengig av at klokkene på de to nodene er synkroniserte. Denne synkroniseringen er imidlertid svært vanskelig å gjøre helt nøyaktig, og det er derfor valgt en annen fremgangsmåte for registrering av de to tidspunktene. I tillegg til hovedprosess og applikasjon, startes det på standbynoden opp en tredje prosess. Denne tar seg av registrering av tidspunkter i forbindelse med takeover og omtales heretter som tidskriverprosessen. Etter at rutinen som skriver

et checkpoint har returnert til den kjørende applikasjonen på primærnoden, sender applikasjonen en melding til tidskriverprosessen. Denne meldingen inneholder et sekvensnummer samt et nummer som representerer nodens identitet. Tidskriverprosessen leser og tolker dataene som mottas i meldingen, beregner tidspunktet ved hjelp av funksjonen *gettimeofday()*, og skriver deretter både nodeidentifikator, sekvensnummer og tidspunkt til en fil, se figur 7.3.

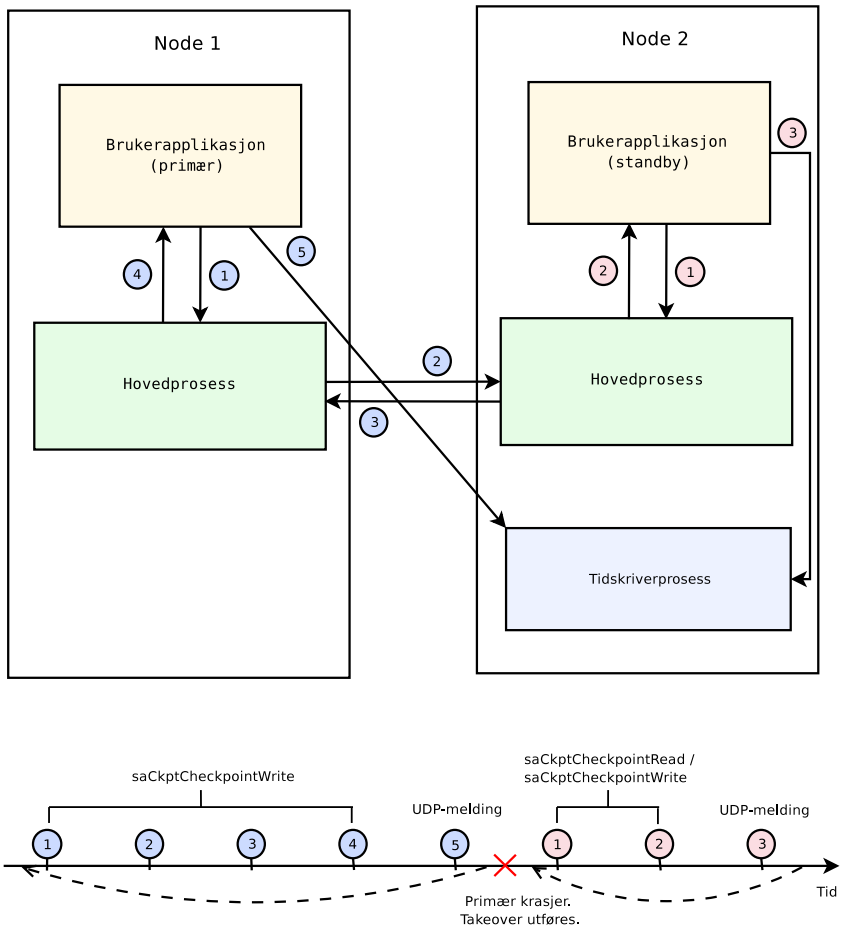
```
1,782,940270620
1,783,940272705
1,784,940283311
1,785,940284846
1,786,940286384
1,787,940288317
1,788,940289846
1,789,940292004
2,790,940881452
2,791,940884555
2,792,940885959
2,793,940888311
2,794,940889538
2,795,940891993
2,796,940906151
```

Figur 7.3: Utdrag fra filen som oppprettes av tidskriverprosessen. Hver linje består av henholdsvis nodenummer, sekvensnummer og tidspunkt.

Ved takeover vil applikasjonen på standbynoden lese inn siste checkpoint og sende en tilsvarende melding idet den er klar til å starte normal utførelse fra det punktet der primærapplikasjonen slapp. Videre vil applikasjonen, på samme måte som sin forgjenger, sende en melding til tidskriverprosessen i etterkant av hver checkpointrutine. Sekvensnummeret som inkluderes i meldingen brukes for å kontrollere at man fortsetter å skrive checkpoints der man slapp etter at en takeover har funnet sted. Nodeidentifikatoren angir hvilken node meldingen stammer fra, og i filen vil endring i nodenummer representere takeover. Takeovertiden beregnes altså som differansen mellom de to tidspunktene som assosieres med endring i nodenummer. I figur 7.3 har takeover funnet sted mellom sekvensnummer 789 og 790.

Hele prosedyren rundt beregning av takeover-tid er illustrert i figur 7.4. Ved å involvere en tredje prosess som tar seg av tidtakingen på denne måten, unngår man altså synkroniseringsproblematikken.

For kommunikasjonen med tidskriverprosessen er det valgt å benytte UDP. Fordelen med å bruke denne teknologien er, særlig i dette tilfellet, at man ikke oppretter noen fast socketforbindelse mellom applikasjonene og tidskriverprosessen. Tidskriverprosessen kan dermed lytte på meldinger fra flere prosesser samtidig. Dette er gunstig da tidtakerprosessen må være rede til å ta imot innkommende meldinger fra både primær- og standbyapplikasjon.



Figur 7.4: Prosedyre for beregning av takeovertid

Kapittel 8

Resultater

I dette kapitlet presenteres resultatene fra testscenariene beskrevet i kapittel 7. Det er foretatt målinger av checkpoint- og takeovertider, og disse illustreres grafisk for å gi en best mulig fremstilling av resultatene. Første delkapittel tar for seg resultatene fra feilfri kjøring og tilhørende målinger av checkpointtider, mens andre delkapittel presenterer resultatene fra kjøring med failover og målinger for takeovertidene. Resultatene kommenteres fortløpende, før konklusjonene trekkes i kapittel 9.

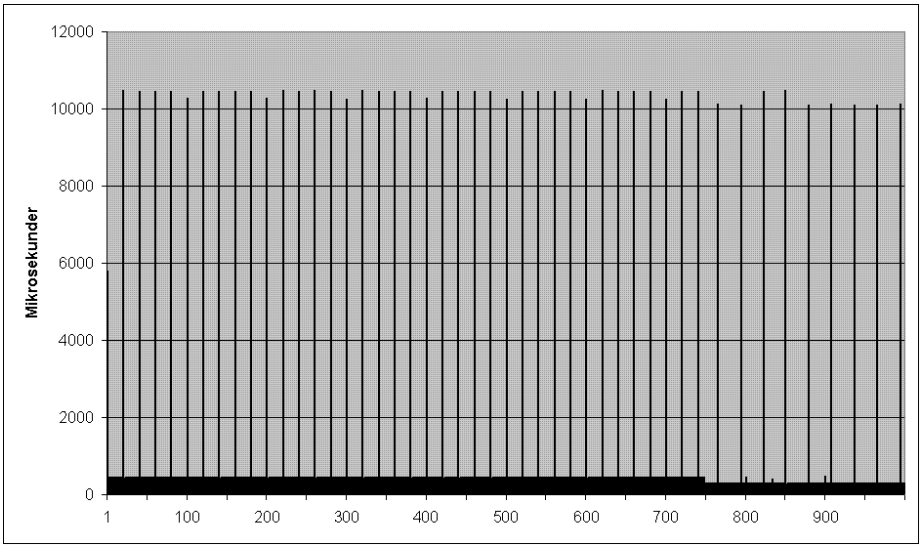
8.1 Feilfri kjøring

Feilfri kjøring innebærer som nevnt i forrige kapittel at systemet oppfører seg som forventet. Det oppstår ingen uventede feil, og det er i dette scenariet kun primærnoden som utfører oppgaver. Primærnoden checkpointer de ulike databufferne, og tiden dette tar måles og skrives til fil. Dette er gjennomført for checkpointstørrelsene beskrevet i kapittel 7.1.

Resultatene fra checkpointtidene til databuffere på størrelser 1 og 10 bytes ga omtrent samme verdier. Av den grunn presenteres kun resultatene fra kjøringer med checkpointstørrelse på 1 byte i denne rapporten.

Figur 8.1 viser resultatet fra 1 000 målinger av skriving av checkpointdata på 1 byte. Det er her en klar tendens til at de fleste checkpointtidene ligger på samme lave nivå, nemlig rundt 450 mikrosekunder. Studerer man grafen nærmere kommer det frem at verdiene synker noe etter omlag 750 kjøringer. Dette skyldes trolig at operativsystemet etter gjentakende operasjoner av samme art selvtunes. Dette betyr at operativsystemet gjenkjenner oppgavene som skal utføres, og vet hvordan ressursene skal utnyttes for å gjenomføre disse mest mulig effektivt. Resultatet blir at utførelsestiden går noe ned.

Det er enkelte verdier i grafen som skiller seg ut ved å være vesentlig høyere enn flertallet, mer presist ligger de i overkant av 10 000 mikrosekunder. Dette kan skyldes *timeslice*n i operativsystemet; en numerisk verdi som angir hvor lenge en prosess kan kjøre før den må vike plass for andre prosesser. Denne variabelen sørger for at det med jevne mellomrom allokeres tid til andre ressurser enn den



Figur 8.1: Tidsmålinger for checkpointstørrelse 1 byte

som er aktiv. Standard *timeslice* på Debian er for tiden 80, 100, 150 eller 200 millisekunder avhengig av hvilken versjon av Linux man kjører. Parametrene HZ eller CLK_TCK angir hvor ofte *scheduler*en sjekker om det er noe annet å gjøre med høyere prioritet eller om det er noen avbrudd som må håndteres. Selv om en prosess ikke har brukt opp sin *timeslice,* kan den bli avbrutt av en prosess med høyere prioritet som er blitt vekket opp av en timer eller en fullført IO-forespørsel. Operativsystemkjernen gjør dessuten en del interne oppgaver ved hvert slikt avbrudd.

En *timeslice* på mellom 80 og 200 millisekunder er svært lang sammenliknet med *timeslice*n i eldre distribusjoner av Linux. Bakgrunnen for dette ligger i prosessorens arkitektur. x86-prosessorene som regjerer på markedet i dag har en meget stor cache (2MB) i forhold til eldre prosessorer. En tråd som har kjørt en stund vil ofte ha skjøvet alt som tilhørte forrige tråd ut av cachen. Når en tråd startes opp igjen vil altså deler av eller hele cachen være erstattet av andre tråders data. Tråden må derfor hente sine data tilbake og ettersom cachen i moderne prosessorer er så stor tar dette relativt mye tid (opptil 10 millisekunder). Med en liten *timeslice* (typisk 10 millisekunder) vil man dermed risikere at en prosess må vike uten å ha gjort stort mer enn å lese inn i cache. For å redusere denne effekten har man har økt *timeslice*n i operativsystemet, slik at oppstartskostnaden blir fordelt over en lengre kjøretid. For å unngå at interaktive oppgaver (det vil si de med mye IO) blir trege, gis disse oppgavene høyere prioritet. Prioriteten justeres dynamisk.

HZ for vår versjon av Debian Linux har verdien 100, det vil si at *scheduler*en gir andre prosesser mulighet til å slippe til hvert 10. millisekund. De høye toppene i resultatet er sannsynligvis en effekt av to tråder med forskjellig prioritet som konkurrerer om CPUen. Da vil man se skedulering med 10 millisekunds intervaller. De høye verdiene innebefatter tiden på 10 millisekunder der en annen

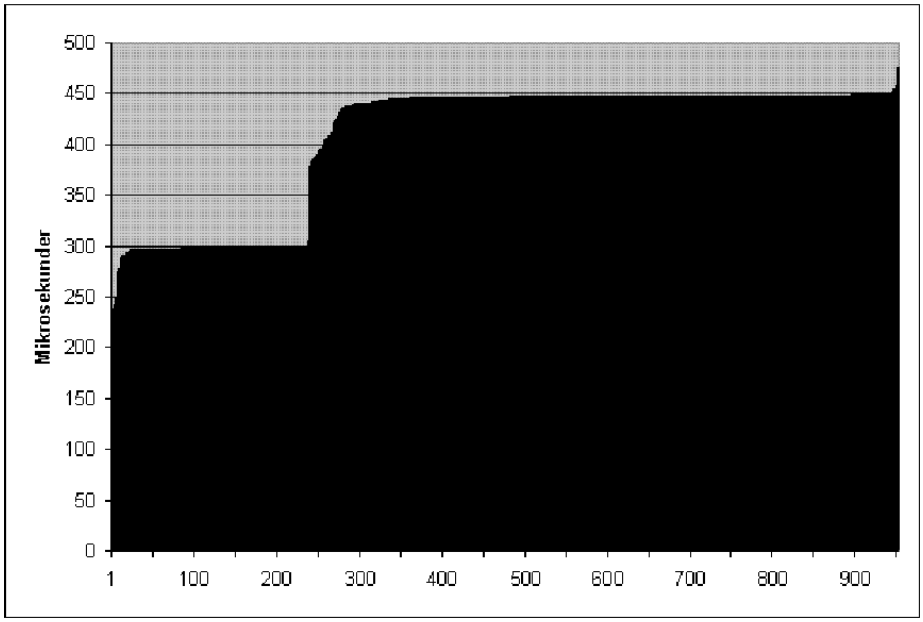
prosess har kommet inn og lagt beslag på CPUen. De jevnt lave verdiene som etterfølger en slik topp summerer opp til ca 10 millisekunder, noe som betyr at vårt program har fått disponere CPUen i dette intervallet. Vi ser at idet tidsforbruket forbundet med hvert checkpoint synker, blir avstanden mellom hver topp større. Altså rekker programmet et større antall checkpoints i løpet av de 10 millisekundene før et kontekstskifte finner sted.

Som et forsøk på å unngå de høye verdiene ble alle andre prosesser enn test-applikasjonen stoppet under testingen. Dette utgjorde imidlertid ingen forskjell i målingene av checkpointtidene da det fremdeles var høye verdier tilstede i grafen. En mulig forklaring på problemet kan være at GUI'et på maskinene bruker endel ressurser, noe det er vanskelig å unngå. Ved å kjøre kommandoen *top*, som indikerer hvilke prosesser som bruker mest ressurser på et gitt tidspunkt, var det kun x-vinduene som ga særlig utslag. Et utsnitt av resultatene fra top-kommandoen er vist i figur 8.2.

PID	USER	PR	NI	VRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1816	root	14	-10	106m	42m	3040	S	1.3	8.3	20:31.74	XFree86
18635	ingunn	12	0	1084	1084	848	R	0.3	0.2	0:00.04	top
1	root	8	0	508	508	456	S	0.0	0.1	0:00.07	init
2	root	9	0	0	0	0	S	0.0	0.0	0:00.16	keventd
3	root	19	19	0	0	0	S	0.0	0.0	0:00.01	ksoftirqd_CPU0
4	root	9	0	0	0	0	S	0.0	0.0	0:00.02	kswapd

Figur 8.2: Utsnitt av resultatet fra top-kommandoen

For å studere de interessante checkpointverdiene fra figur 8.1 nærmere, ble verdiene målt til over 10 000 mikrosekunder fjernet. De resterende resultatene ble så sortert etter økende tid, noe som er illustrert i figur 8.3.

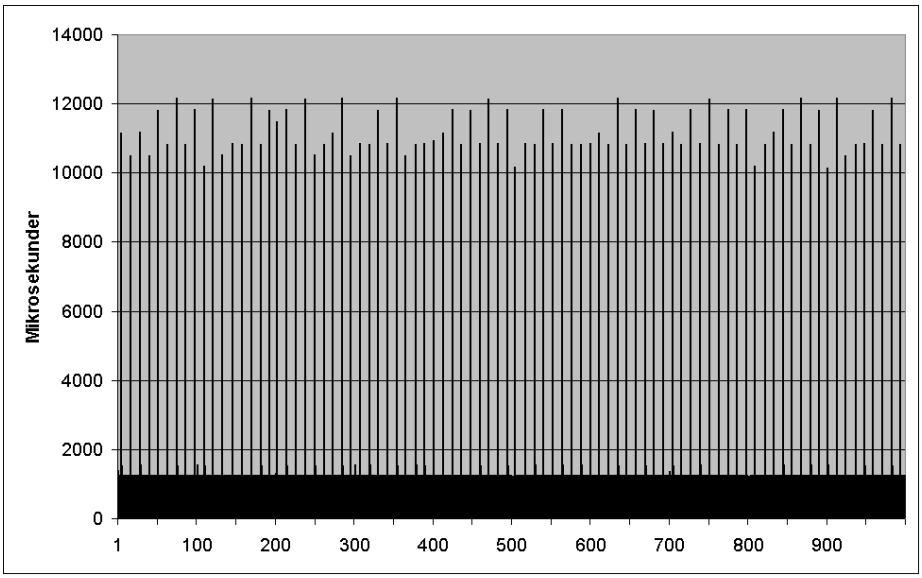


Figur 8.3: Sorterte resultater for checkpointstørrelse 1 byte

I figuren kommer det tydelig frem at checkpointtidene ligger på to ulike nivåer. Det høyeste nivået er checkpointtidene som opptrer i starten av testkjøringen, mens det laveste nivået sannsynligvis er forårsaket av selvtuningen i operativsystemet. Verdiene på disse nivåene ligger på rundt henholdsvis 450 og 300 mikrosekunder. Selvtuningen i operativsystemet gir altså en nedgang i checkpointtiden på ca 35%, noe som er en relativt stor forbedring. Dersom det hadde vært utført flere iterasjoner ville sannsynligvis de resterende checkpointtidene plassert seg på det laveste nivået. Gjennomsnittstiden ville dermed blitt lavere jo flere iterasjoner som ble utført.

Resultatene fra checkpointtidene av databuffere på størrelsene 100 og 1 000 bytes ga omtrent like store utslag. Det blir derfor kun presentert resultater fra kjøringer med checkpointstørrelse på 100 bytes i denne rapporten.

Figur 8.4 viser resultatet fra 1 000 målinger av skriving av checkpointdata på 100 bytes. Vi ser her den samme tendensen som i forrige tilfelle, nemlig at det forekommer høye verdier med jevne mellomrom. Dette skyldes igjen *scheduler*en i operativsystemet som gjør at andre prosesser får komme til. En annen observasjon som gjøres med denne checkpointstørrelsen, selv om det ikke kommer frem av figuren, er at annenhver verdi er lav og høy - de ligger rundt 300 og 1 300 mikrosekunder. Det samme er tilfellet ved checkpointdata på 1 000 bytes, men fenomenet gjentok seg ikke for noen av de andre checkpointstørrelsene.

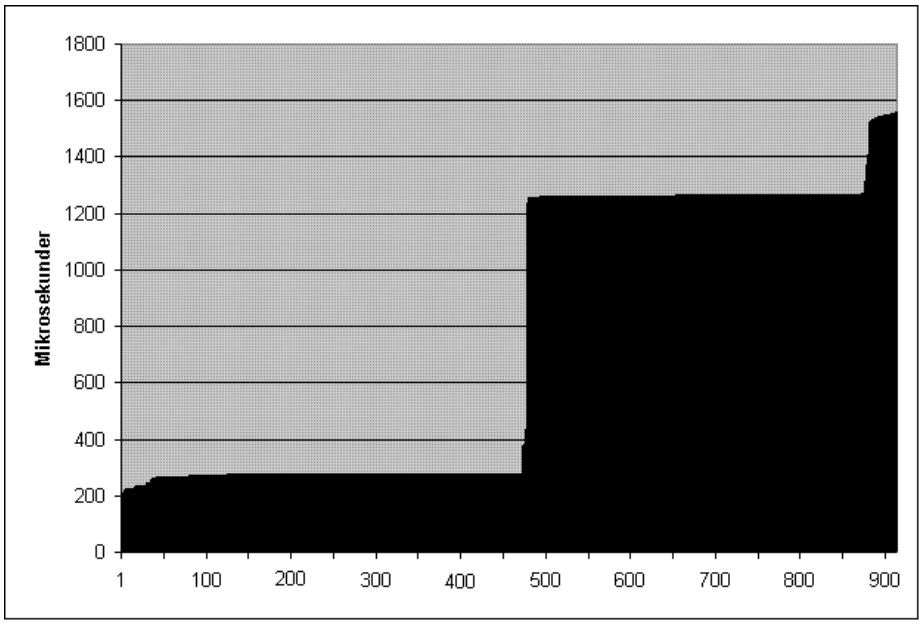


Figur 8.4: Tidsmålinger for checkpointstørrelse 100 bytes

Som et forsøk på å jevne ut verdiene ble en TCP/IP-variabel kalt TCP_NODELAY satt. Denne skal sørge for at pakkene sendes over til mottakersiden umiddelbart. Siden det er ineffektivt å sende små pakker, prøver TCP-implementeringen i Linux å aggregere dataene før de sendes ut. Et eksempel på dette

er noe kalt Nagels regel, hvor halvfulle pakker kan gis en kort forsinkelse før de sendes ut i håp om at mer data vil bli tilgjengelig. Slike forsinkelser kan forøvrig påvirke ytelsen i programmet da man venter på at pakkene skal fylles opp med data før de sendes. Å sette variabelen TCP_NODELAY ga imidlertid ikke utslag på resultatene. I [23] beskrives en undersøkelse som konkluderer med at TCP_NODELAY ikke hjelper på små pakkestørrelser. Det er imidlertid utviklet en *patch* som kan kompileres inn i operativsystemkjernen for å forbedre tidsaspektet ved oversendelse av pakker. Denne patchen ble ikke testet ut i dette prosjektet, og det er derfor uvisst om det ville hatt noen effekt på de utførte målingene.

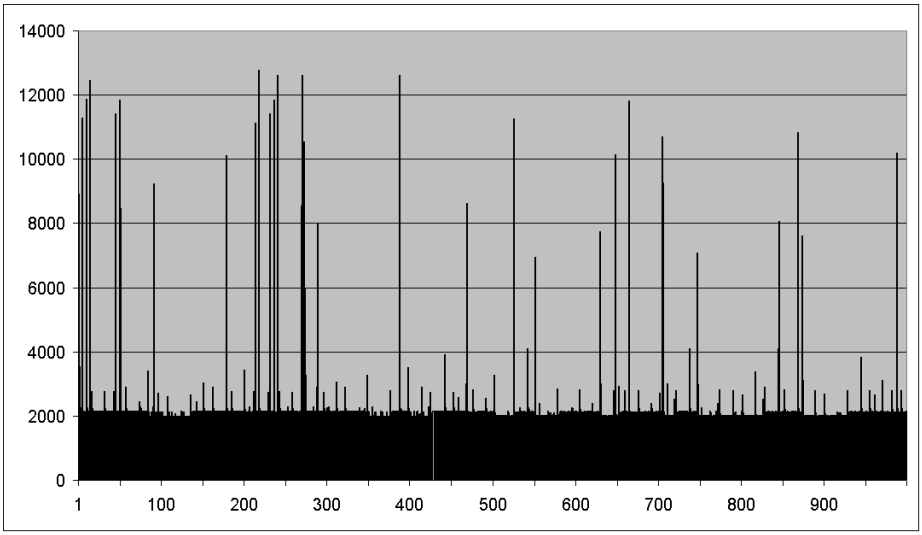
For enklere å studere de interessante checkpointtidene, ble igjen de høyeste verdiene på målingene fjernet. Resultatene ble så sortert etter økende tid, noe som er vist i figur 8.5. Her ser man tydelig de to ulike platåene checkpointtidene ligger på. Nivåene isolert sett er meget jevne, men forskjellen mellom verdiene de representerer er veldig stor. Det er interessant å merke seg at verdien på det laveste nivået i dette tilfellet tilsvarer nivået etter selvtuning i tilfellet med checkpointstørrelse på 1 byte (se figur 8.1 og 8.3).



Figur 8.5: Sorterte resultater for checkpointstørrelse 100 bytes

Det ble også kjørt målinger på checkpointtidene til et databuffer på 10 000 bytes. Resultatet fra disse testkjøringene er vist i figur 8.6.

Det forekommer også her verdier som er betydelig høyere enn flertallet, men disse opptrer ikke jevnt fordelt over perioden som i de andre tilfellene. Disse høye verdiene er heller ikke like jevne i størrelse som for de andre checkpointstørrelsene, men fordeler seg mellom 7 000 og 13 000 mikrosekunder. Hovedtyngden av tidsmålingene ligger rundt 2 000 mikrosekunder, noe som er en høyere verdi enn for testkjøringene med 1 og 100 bytes store databuffere. Dette



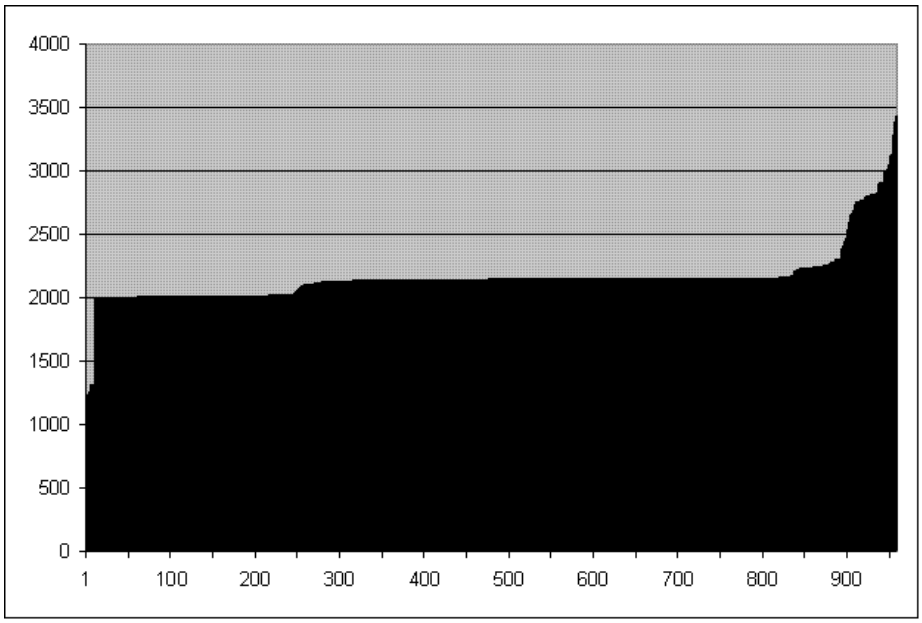
Figur 8.6: Tidsmålinger for checkpointstørrelse 10 000 bytes

er forventet oppførsel da det er mer data som skal skrives og overføres i dette tilfellet, og det følgelig tar lengre tid å utføre checkpointingen.

Dersom de høyeste verdiene fra tidsmålingene av checkpointstørrelser på 10 000 bytes fjernes og de resterende resultatene sorteres, får man grafen som er vist i figur 8.7. Også her ser vi at resultatene befinner seg på to ulike platåer, men forekomsten av verdier på de to platåene opptrer spredt utover på de gjennomførte iterasjonene. Det ser dermed ikke ut til å skyldes at operativsystemet selvtuner og oppførselen er av uvisst opphav.

Som indikert i kapittel 7.1 var det også ønskelig å måle hvor mye CPU-ressurser som ble brukt under checkpointing av ulike bufferstørrelser. Til dette skulle funksjonskallet *getrusage()* benyttes. Det viste seg imidlertid, etter gjentatte forsøk, at funksjonen ikke fungerte som forventet. Verdiene som ble returnert var veldig varierende, og av en slik natur at de ikke virket pålitelige. Granulariteten fremstod meget høy, og målinger for ett enkelt checkpoint returnerte 0 CPU forbruk. For å fremprovosere et resultat ble det derfor foretatt målinger av totalt CPU-forbruk for skriving av 10 000 checkpoints, men også her ble resultatet i enkelte tilfeller 0. På flere internettsider og diskusjonsforum finnes også indikasjoner på at *getrusage()* ikke fungerer korrekt på Linux. I denne rapporten ble det derfor valgt å ikke gjengi testresultatene fra kjøring av denne funksjonen da de ikke ga noe fornuftig grunnlag for konklusjoner vedrørende CPU-forbruket.

For å allikevel få en viss indikasjon på hvor stor andel av CPU-ressursene applikasjonen opptok, ble kommandoen *vmstat* benyttet. Denne viser statistikk over virtuelt minneforbruk, og ble stilt inn til å gi oppdatert informasjon hvert sekund. Det er to av parametrene som er spesielt interessante i denne analysen, nemlig us (*user time*) og sy (*system time*). Dette er de samme parameterne som *getrusage()* returnerer informasjon om - *user time* er tiden brukt på å kjøre



Figur 8.7: Sorterte resultater for checkpointstørrelse 10 000 bytes

brukerimplementert programkode, mens *system time* er tiden brukt på systemkall til kjernen. Forskjellen mellom *getrusage()* og *vmstat* er at *vmstat* returnerer prosentandelen av den totale CPU-tiden som er brukt, ikke den konkrete tidsangivelsen for hver prosess. Opplysningene vil allikevel være nyttige for å få en indikasjon på hvor stor differansen i CPU-forbruk er mellom skriving av de ulike checkpointstørrelsene.

```
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 4  0   1592  44760  91384 155716    0    0     0     0  227   795  5  0 95  0
 3  0   1592  44760  91384 155716    0    0     0     0  232   793  4  1 95  0
 2  0   1592  44752  91384 155716    0    0     0    96  238   788  5  0 95  0
 1  0   1592  44752  91384 155716    0    0     0     0  227   747  5  0 95  0
 2  0   1592  44752  91384 155716    0    0     0     0  229   776  5  0 95  0
 1  0   1592  44752  91384 155716    0    0     0     0  228   788  5  0 95  0
 3  0   1592  44748  91388 155716    0    0     0    80  232   780  5  0 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  231   764  5  0 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  226   788  5  0 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  231   806  4  1 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  228   794  5  0 95  0
 3  0   1592  44748  91388 155716    0    0     0     0  227   774  5  0 95  0
 2  0   1592  44748  91388 155716    0    0     0     0  227   788  5  1 94  0
 1  0   1592  44748  91388 155716    0    0     0     0  229   795  5  0 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  227   802  5  0 95  0
 1  0   1592  44748  91388 155716    0    0     0     0  226   788  5  0 95  0
 3  0   1592  44740  91396 155716    0    0     0    56  232   780  5  0 95  0
```

Figur 8.8: Resultater fra kommandoen *vmstat* når systemet er i initiell tilstand

Først ble *vmstat*-kommandoen kjørt uten noen aktive ressurser på maskinen for å få en oversikt over hvor store CPU-andeler som forbrukes i en slik initiell tilstand. Disse verdiene kan trekkes fra prosentverdiene som observeres under kjøring av applikasjonen da de initielle prosessene alltid kjører i bakgrunnen og CPU-forbruket av disse er konstant. Resultatet fra kjøring av *vmstat*-kommandoen med en samplingsrate på 1 sekund i systemets initielle tilstand er vist i figur 8.8.

Kolonne 3 og 4 fra høyre viser henholdsvis prosentvis andel av *system time* og *user time* som forbrukes når maskinen er i en initiell tilstand. *user time* ligger her på verdiene 4 og 5, mens *system time* har verdiene 0 og 1. Gjennom eksperimenter viste det seg at størrelsen på x-vinduet som *vmstat* kjørte i var avgjørende for hvor mye CPU-kraft som bruktes. Ved å vise terminalvinduet over hele skjermen ble ressursforbruket doblet i forhold til det figur 8.8 viser, mens en minimering av terminalvinduet førte til at både *user time* og *system time* ble 0. Dette tilsier at ressursforbruket i initiell tilstand til enhver tid avhenger av hvor stor del av skjermbildet som må oppdateres og med hvilken frekvens. Så lenge *vmstat* kjøres med konstant samplingsrate og i et vindu med konstant størrelse, vil verdiene for ressursforbruk kunne sammenliknes fra kjøring til kjøring. Verdiene i figur 8.8 kan dermed brukes som et utgangspunkt for initiell tilstand og subtraheres fra de tilsvarende verdiene som vises under kjøring av testapplikasjonen, for å estimere ressursforbruk forbundet checkpointing.

```
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 6  0   1592  58264  91896 166532    0    0     0     0 1314  4928  8  1 91  0
 2  0   1592  58200  91896 166532    0    0     0     0 1303  4885  8  3 89  0
 3  0   1592  58136  91896 166532    0    0     0     0 1271  4746  7  5 88  0
 2  0   1592  58072  91896 166532    0    0     0     0 1287  4813  5  6 89  0
 2  0   1592  58004  91896 166532    0    0     0     0 1305  4908  7  7 86  0
 3  0   1592  57936  91896 166532    0    0     0     0 1321  4960  8  5 87  0
 2  0   1592  57868  91896 166532    0    0     0     0 1344  5048  6  4 90  0
 1  0   1592  57800  91896 166532    0    0     0     0 1352  5063  6  7 87  0
 2  0   1592  57732  91896 166532    0    0     0     0 1338  5059  8  6 86  0
 1  0   1592  57664  91896 166532    0    0     0     0 1348  5049  7  2 91  0
 4  0   1592  57592  91896 166532    0    0     0     0 1384  5220  8  4 88  0
 2  0   1592  57524  91896 166532    0    0     0     0 1401  5295  8  2 90  0
 3  0   1592  57448  91896 166532    0    0     0     0 1480  5624  7  5 88  0
 3  0   1592  57372  91896 166532    0    0     0     0 1457  5536 10  2 88  0
 2  0   1592  57296  91896 166532    0    0     0     0 1465  5560  7  5 88  0
 4  0   1592  57220  91896 166532    0    0     0     0 1434  5446  7  3 90  0
 2  0   1592  57144  91896 166532    0    0     0     0 1463  5584  4  0 96  0
```

Figur 8.9: Resultater fra kommandoen *vmstat* under checkpointing av 1 byte

Kommandoen *vmstat* ble videre kjørt samtidig som testprogrammet kontinuerlig utførte checkpointing av et buffer på 1 byte. Resultatet er vist i figur 8.9. Her ser man tydelig at *user time* og *system time* utgjør høyere prosentandeler enn i forrige tilfelle. *user time* ligger nå hovedsaklig på 7 og 8, noe som angir et totalt CPU-forbruk på ca 3% for applikasjonen under checkpointing av buffer på 1 byte. *system time* er noe varierende, men har hovedtyngde på verdier mellom 4 og 7. Det er ønskelig å sammenlikne dette med CPU-forbruket til checkpointing av buffer på 10 000 bytes. På forhånd var det forventet at *user*

time skulle ligge noe høyere for 10 000 bytes, da det er et større buffer som skal håndteres. *system time* var forventet å ligge på samme nivå siden antall systemkall som foretas er det samme uavhengig av checkpointstørrelsen. Figur 8.10 viser resultatet fra kommandoen *vmstat* med samplingsrate på 1 sekund utført mens det kontinuerlig checkpointes verdier på 10 000 bytes.

```
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 3  0   1592  64468  91796 165988    0    0     0     0 1827  3073  9  5 86  0
 3  0   1592  64436  91796 165988    0    0     0     0 1849  3084  7  3 90  0
 4  0   1592  64416  91796 165988    0    0     0     0 1824  3061  7  5 88  0
 2  0   1592  64388  91796 165988    0    0     0     0 1837  3069  9  2 89  0
 3  0   1592  64372  91796 165988    0    0     0     0 1876  3111  9  6 85  0
 3  0   1592  64336  91796 165988    0    0     0     0 1877  3145 11  5 84  0
 3  0   1592  64320  91796 165988    0    0     0     0 1850  3086  7  2 91  0
 2  0   1592  64292  91796 165988    0    0     0     0 1752  2937  8  6 86  0
 2  0   1592  64272  91796 165988    0    0     0     0 1858  3103  6  2 92  0
 5  0   1592  64240  91796 165988    0    0     0     0 1807  3035  9  5 86  0
 2  0   1592  64220  91796 165988    0    0     0     0 1822  3042  7  2 91  0
 2  0   1592  64200  91796 165988    0    0     0     0 1849  3073  7  7 86  0
 2  0   1592  64168  91796 165988    0    0     0     0 1851  3095 10  2 88  0
 2  0   1592  64148  91796 165988    0    0     0     0 1762  2951 11  4 85  0
 4  0   1592  64124  91796 165988    0    0     0     0 1883  3135  7  6 87  0
 2  0   1592  64104  91796 165988    0    0     0     0 1793  3000 10  7 83  0
 3  0   1592  64080  91796 165988    0    0     0     0 1687  2848  7  7 86  0
```

Figur 8.10: Resultater fra kommandoen *vmstat* under checkpointing av 10 000 bytes

Verdiene på *user time* varierer her fra 6 til 11, med hovedtyngden mellom 7 og 9. Dette er noe høyere enn for checkpointing av 1 byte. Det forekommer også verdier på 10 og 11, noe som indikerer et totalt CPU-forbruk på opp mot 6% for applikasjonen under checkpointing av buffer på 10 000 bytes. Flesteparten av verdiene tilsier at applikasjonen forbruker 4% av de totale CPU-ressursene, noe som ikke er særlig høyere enn for checkpointing av 1 byte. Resultatene indikerer allikevel at det krever mer CPU-kraft å prosessere checkpointing av større buffere. *system time* er også noe varierende i dette tilfellet, med hovedtyngde på verdiene 5 og 6. Det er ingen indikasjon på at *system time* er høyere for checkpointing av større verdier, noe som også var et forventet resultat.

Estimatene beskrevet ovenfor viser at den implementerte løsningen holder ressursforbruket på et lavt nivå selv om det er relativt store mengder informasjon som checkpointes. Dette betyr at overhead forbundet med checkpointing ved feilfri kjøring er lav både når det gjelder tidsforbruk og ytelse. Systemet vil dermed ha stor kapasitet ledig til å la applikasjonen utføre sine primære oppgaver.

8.2 Kjøring med failover

Det er tre ulike scenarier som er testet i forbindelse med failover; at applikasjonen feiler, at hovedprosessen feiler og at noden applikasjonen kjører på krasjer. Alle disse feilsituasjonene ble fremprovosert av testerne. Utførelsen av testene ble repetert gjentatte ganger med checkpointdata på størrelsene 1 byte og 10 000

bytes. Det viste seg etter gjennomføring av testene at størrelsen på checkpoint-dataene hadde svært liten effekt på takeovertiden. Av den grunn presenteres hovedsaklig resultatene for en checkpointstørrelse på 1 byte.

På forhånd var det forventet at scenariene hvor applikasjonen feiler og hvor hovedprosessen feiler ville få lavere takeovertider enn ved nodekrasj. Dette er fordi noden fortsatt er aktiv i de to første tilfellene, og dermed kan kommunisere med standbynoden over TCP-forbindelsen. Dette er forventet å gi raskere utførelse enn når primærnoden går ned. I sistnevnte tilfelle må Heartbeat detektere at det har oppstått et krasj, og deretter gi beskjed til standbynoden om å foreta en takeover. En nærmere beskrivelse av de ulike scenariene for deteksjon av feil er beskrevet i påfølgende underkapitler.

8.2.1 Applikasjonen feiler

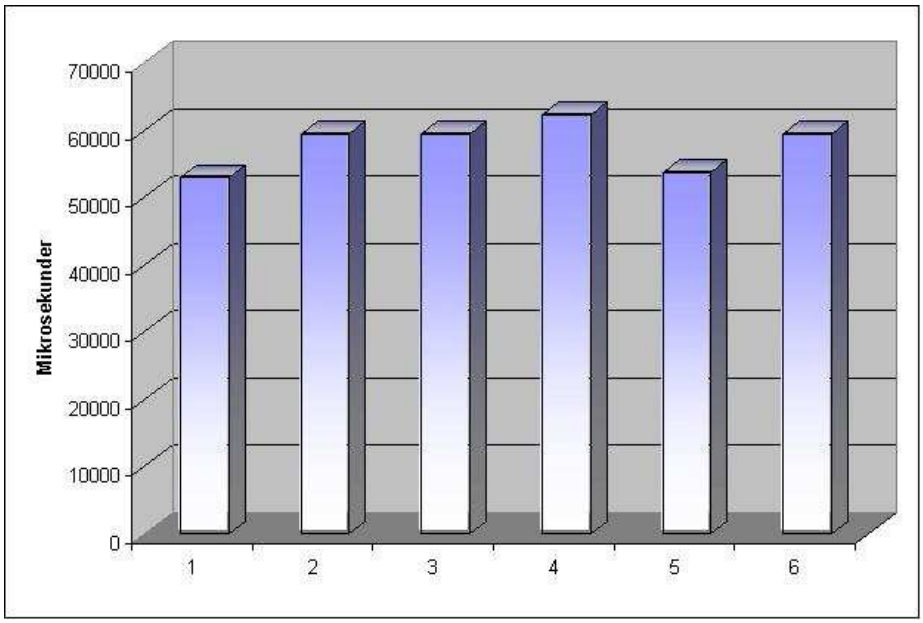
For å simulere en applikasjonsfeil ble programmet krasjet manuelt ved å drepe prosessen. Når applikasjonen feiler detekteres dette av hovedprosessen på den aktuelle noden, som sender beskjed til hovedprosessen på standbynoden om at den skal ta over kontrollen. Tiden det tar fra primærnoden skriver sitt siste checkpoint til standbynoden har startet opp og lest denne checkpointverdien, måles og lagres i en fil sammen med nodenummer og sekvensnummer (se kapittel 7.2). Resultatet fra et tilfeldig utvalg kjøringer hvor en slik feil er simulert er vist i figur 8.11. På x-aksen er nummeret på den aktuelle kjøringen angitt, mens y-aksen viser takeovertiden i mikrosekunder.

Overtakelsestiden er her meget rask, noe som i stor grad skyldes TCP/IP forbindelsen. Hovedprosessen oppdager at forbindelsen til applikasjonen, som er lokalisert på samme node, er mistet og sender deretter beskjed til standbynodens hovedprosess om å starte opp. Dette er mulig da noden fremdeles er i live og kommunikasjonskanalen til standbynoden fortsatt eksisterer. Kommunikasjon via TCP/IP forbindelser går meget raskt, og dette bidrar til lave takeovertider.

Det er også tydelig at variasjonen på målingene er relativt liten. De strekker seg fra 53 023 til 62 327 mikrosekunder, noe som gir en forskjell på 17,5% i verdi. Årsaken til dette er at TCP/IP er en stabil og velutviklet protokoll. I tillegg er det meget små mengder data som sendes via kommunikasjonskanalene, noe som medvirker til stabiliteten av tjenesten. Sendes store mengder av data øker sjansene for større variasjon i målingene.

8.2.2 Hovedprosessen feiler

For å simulere feil i hovedprosessen ble denne krasjet manuelt ved å drepe prosessen. At hovedprosessen feiler på primærnoden detekteres av hovedprosessen på standbynoden, da den merker at forbindelsen til hovedprosessen på primærnoden er mistet. Når dette oppdages starter hovedprosessen på standbynoden opp applikasjonen på sin node, og fortsetter deretter utførelsen der primærnoden krasjet. Tiden det tar fra primærnoden skriver sitt siste checkpoint til standbynoden har startet opp og lest denne checkpointverdien, lagres i en fil sammen med nodenummer og sekvensnummer (se kapittel 7.2). Resultatene fra et tilfeldig utvalg kjøringer hvor en slik feil er simulert er vist i figur 8.12. På



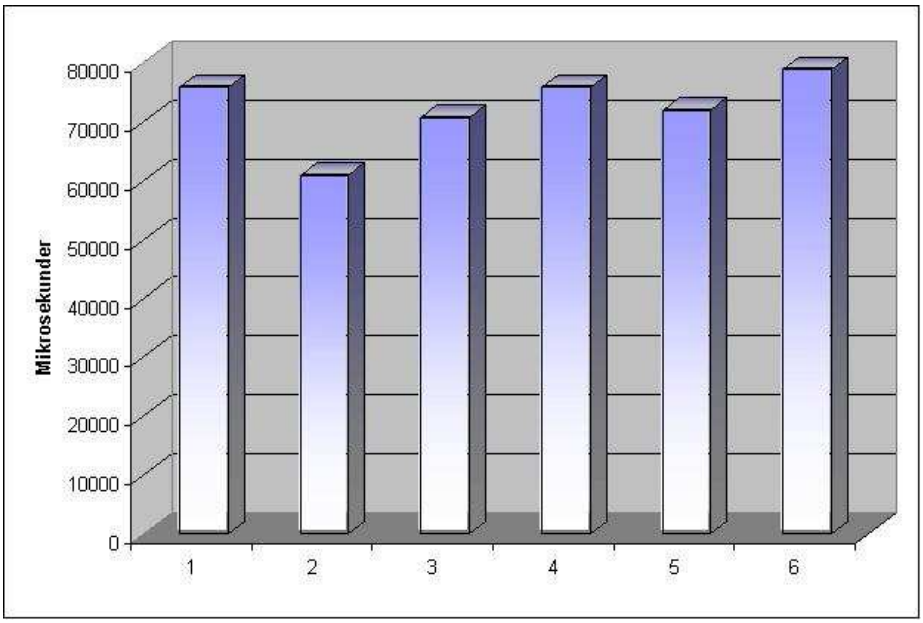
Kjøring 1	Kjøring 2	Kjøring 3	Kjøring 4	Kjøring 5	Kjøring 6
53023	59487	59522	62372	53769	59482

Figur 8.11: Målinger av takeovertider når applikasjonen feiler

x-aksen er nummeret på den aktuelle kjøringen angitt, mens y-aksen viser takeovertiden i mikrosekunder.

Takeovertiden er også her veldig lav, men ligger dog noe høyere enn i forrige scenario. Dette skyldes at det nå er TCP/IP forbindelsen mellom nodene som må detektere at en av endeapplikasjonene har gått ned. Dette tar litt lengre tid enn når endepunktene i forbindelsen er lokalisert på samme node. Årsaken er at når endepunktene er plassert i samme datastruktur trenger det ikke gå trafikk på forbindelsen for å detektere at en av enhetene har gått ned. Når endepunktene derimot er plassert i ulike datastrukturer, må det gå en pakke mellom disse for å indikere at en av endepunktene er nede. Det vil i utgangspunktet kun gå trafikk på linjen når det er data å sende, noe det ikke vil være etter et eventuelt krasj. Dersom det etter en viss tidsperiode ikke har ankommet pakker til mottakerapplikasjonen, vil denne sende en forespørsel til avsenderapplikasjonen for å innhente informasjon om dens status. Siden noden denne kjører på fremdeles er i live, vil det bli returnert svar om at applikasjonen det sendes forespørsel om har gått ned. Hele denne prosessen vil ta noe lenger tid enn å detektere krasjede endeappliksjoner på samme node.

Variasjonen på målingene er litt større enn ved forrige scenario. De strekker seg fra 61 001 til 78 948 mikrosekunder, noe som gir en forskjell på 29,4% i verdi. Ser man forøvrig bort fra den laveste verdien, som avviker mest fra de andre verdiene, blir verdiforskjellen på 11,6%. Dette er igjen en relativ liten spredning, og kan forklares med stabiliteten i TCP/IP.



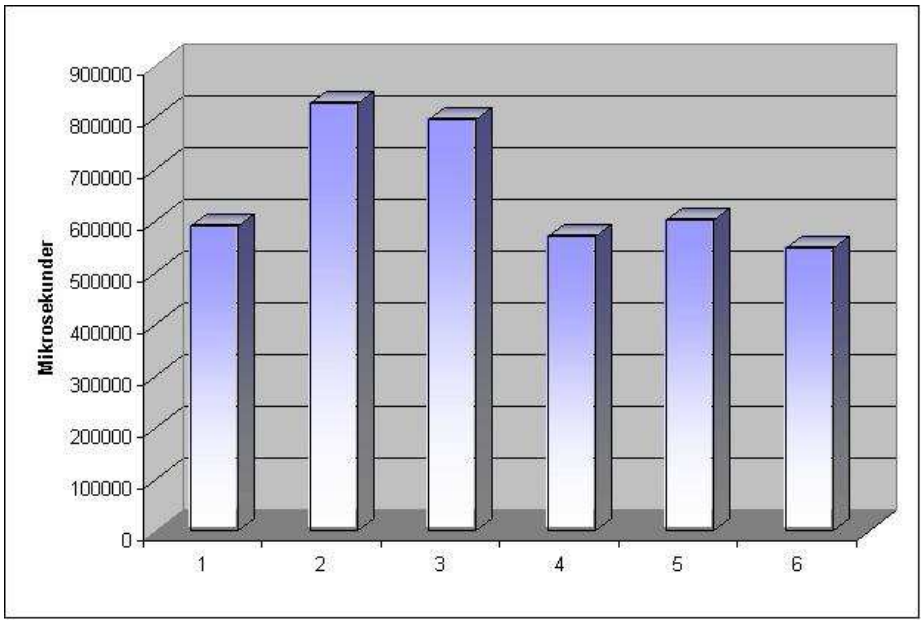
Kjøring 1	Kjøring 2	Kjøring 3	Kjøring 4	Kjøring 5	Kjøring 6
75945	61001	70758	75954	71969	78948

Figur 8.12: Målinger av takeovertider når hovedprosessen feiler

8.2.3 Noden applikasjonen kjører på krasjer

Dersom primærnoden krasjer skal dette detekteres av Heartbeat. Det sendes kontinuerlig heartbeats fra primærnoden for å indikere at den er i live. Dersom disse opphører, gir Heartbeat beskjed til standbynoden så snart krasjet er oppdaget og en viss tidsperiode kalt *deadtime* har passert. *deadtime* angir hvor lenge man skal vente før en node erklæres død. Dette er for å sikre at noden virkelig har gått ned, og at uteblitte heartbeats ikke skyldes andre feil som for eksempel mistede pakker. Når standbynoden får beskjed fra Heartbeat om å ta over for primærnoden leser den siste checkpointede verdi som ble lagret, og fortsetter utførelsen derfra.

For å simulerere et nodekrasj kan man for eksempel fjerne all strømtilførsel til maskinen ved å trekke ut støpselet fra veggen. Da dette er en meget drastisk operasjon, som kan medføre skade på maskinen og programvaren, ble det i stedet valgt å fjerne nettilgangen ved å trekke ut nettverkskabelen til maskinen. Dette vil simulere et krasj overfor Heartbeat og de andre nodene i clusteret. Heartbeat vil tolke dette som at noden har gått ned, da det ikke lenger er mulig å kommunisere med den og det ikke mottas heartbeats i løpet av *deadtime* perioden. Tiden fra siste checkpoint blir skrevet av primærnoden til standbynoden leser denne verdien, skrives til en fil sammen med nodenummer og sekvensnummer (se kapittel 7.2). Et utvalg av resultatene hvor denne feilsituasjonen er simulert er vist i figur 8.13.



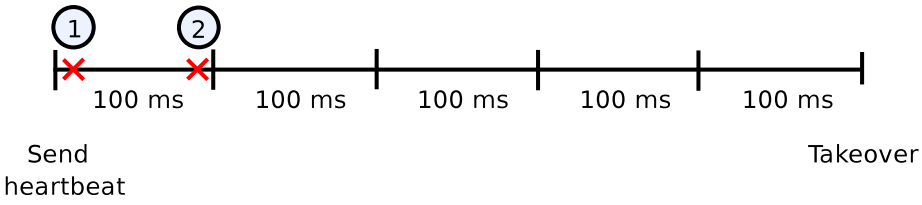
Kjøring 1	Kjøring 2	Kjøring 3	Kjøring 4	Kjøring 5	Kjøring 6
589448	828770	797470	590628	602432	547908

Figur 8.13: Målinger av takeovertider når primærnoden krasjer

Verdien på takeovertidene er, som forventet, mye høyere for dette scenariet enn for de to andre alternativene. Grunnen til at verdiene er høyere skyldes at det her er Heartbeat som må detektere at noden har gått ned. I de andre tilfellene, hvor prosessene ble drept og noden fortsatt var i live, ble det oppdaget av TCP/IP at forbindelsen til applikasjonen var tapt. I dette scenariet kan ikke TCP/IP benyttes til å detektere at endeapplikasjonen har gått ned. Protokollen vil vente på svar fra den krasjede noden angående status på den feilede applikasjonen til noden eventuelt kommer opp igjen. Det overlates derfor til Heartbeat å detektere feilen, da denne innhenter informasjon om hvilke noder som er oppe og nede.

Som det vises i figuren er variasjonen på målingene relativt stor i dette tilfellet. Den strekker seg fra 547 908 til 828 770 mikrosekunder, noe som gir en forskjell på 51,3% i verdi. Den store spredningen kan blant annet skyldes en konfigurasjonsparameter som Heartbeat benytter, nemlig *keepalive*. Denne angir hvor ofte heartbeats skal sendes fra primærnoden for å indikere at den er i live. Dersom noden krasjer rett etter at et heartbeat er sendt, må hele denne *keepalive*-perioden ventes før Heartbeat vil "savne" en melding fra noden. Krasjer noden derimot rett før et Heartbeat skal sendes, vil ikke dette utgjøre noen tidsforsinkelse. Dette er nærmere illustrert i figur 8.14. Her er *keepalive* satt til 100 millisekunder, og det antas at 5 pakker kan mistes før primærnoden erklæres død og standbynoden skal overta prosesseringen.

I tilfelle 1 har det oppstått et krasj rett etter et heartbeat ble sendt. Det vil da ta opp mot 100 millisekunder før et slikt signal savnes av Heartbeat. I tilfelle 2

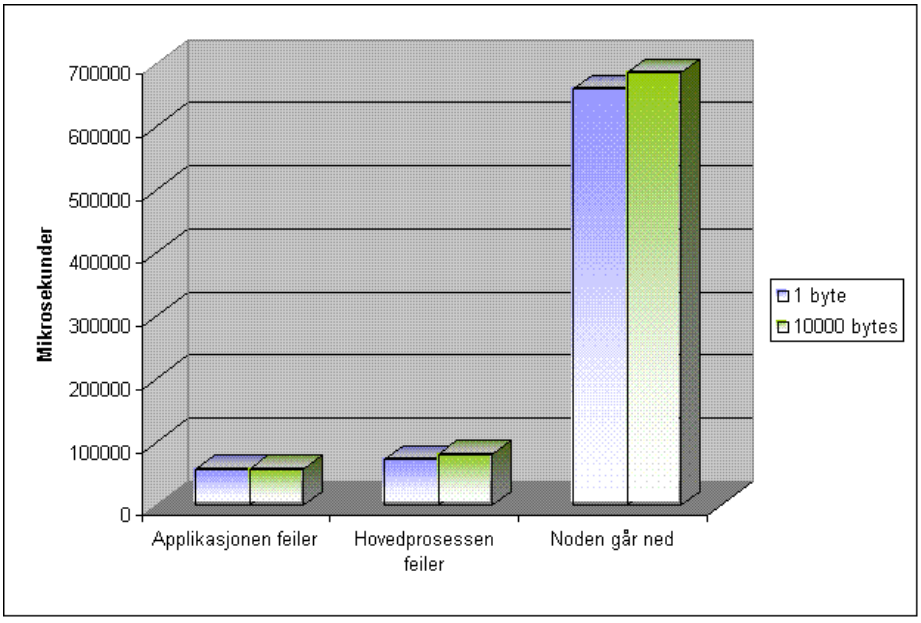


Figur 8.14: Variasjon i takeovertid ettersom når nodekrasjet opptrer

opptrer krasjet rett før et heartbeat skal sendes. I dette tilfellet vil det ta ned mot 0 millisekunder før signalet savnes. I et verste tilfelle vil tidsforskjellen på to uavhengige feilsituasjoner som er simulert under testing bli opp mot 100 millisekunder, avhengig av når krasjet oppsto. Dette kan forklare noe av spredningen som opptrer i målingene.

8.2.4 Sammenstilling av resultater og tuning av Heartbeat

Figur 8.15 viser forskjellene i takeovertid mellom de tre scenariene. Her er det tatt med et gjennomsnitt av kjøringene med checkpointstørrelser på 1 byte og 10 000 bytes. Det kommer helt klart frem at de to første alternativene er betydelig raskere med tanke på takeover. I tillegg viser diagrammet at forskjellen i takeovertid for checkpointstørrelser på 1 byte og 10 000 bytes er relativt liten.



Figur 8.15: Målinger av takeovertider for alle metodene

Siden takeovertiden ved nodekrasj avvek vesentlig fra de andre to scenariene,

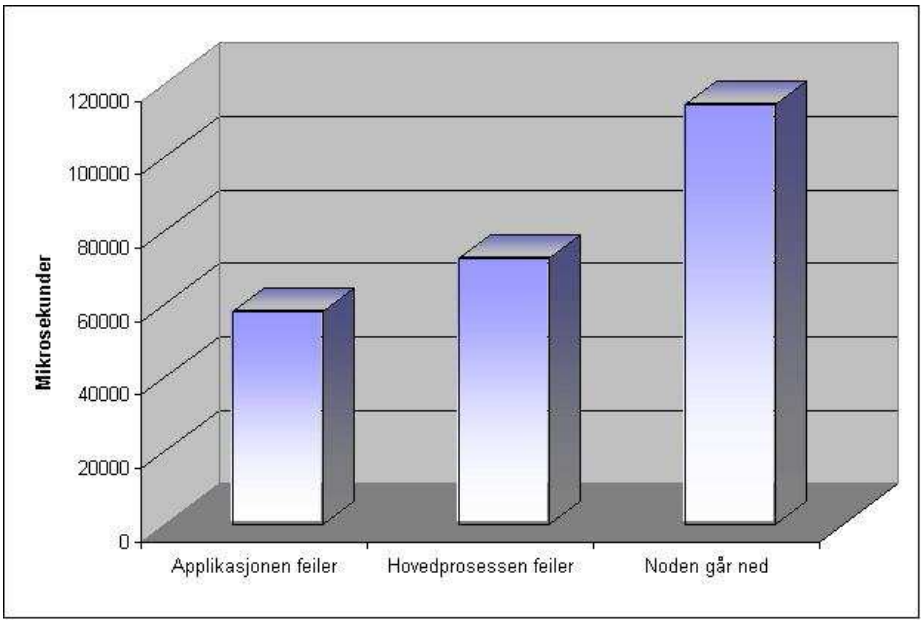
var det interessant å se etter metoder for å redusere tidsforbruket for dette tilfellet. I konfigurasjonsfilen til Heartbeat, som leses idet programmet starter opp, finnes et antall justerbare parametre som har innvirkning på takeovertiden. Dette gjelder særlig parametrene *deadtime* og *keepalive* som er nevnt over. *deadtime* sier hvor lang tid det skal gå før man erklærer primærnoden død. *keepalive* er tiden mellom hvert heartbeat som sendes fra primær til standby for å indikere at primærnoden er i live. Antall mistede heartbeats som kreves for at en node skal erklæres død vil da bli (*deadtime*/*keepalive*). Et utsnitt av konfigurasjonsfilen er vist i figur 8.16.

```
#       keepalive: how long between heartbeats?
#
keepalive 10ms
#
#       deadtime: how long-to-declare-host-dead?
#
#               If you set this too low you will get the problematic
#               split-brain (or cluster partition) problem.
#               See the FAQ for how to use warntime to tune deadtime.
#
deadtime 80ms
#
#       warntime: how long before issuing "late heartbeat" warning?
#       See the FAQ for how to use warntime to tune deadtime.
#
warntime 40ms
#
#       Very first dead time (initdead)
#
#       On some machines/OSes, etc. the network takes a while to come up
#       and start working right after you've been rebooted.  As a result
#       we have a separate dead time for when things first come up.
#       It should be at least twice the normal dead time.
#
initdead 10
#
```

Figur 8.16: Utsnitt av konfigurasjonsfilen til Heartbeat

Det ble utført tester på hvordan takeovertiden ble påvirket av endringer i disse parametrene. På forhånd virket det sannsynlig at en reduksjon i både *deadtime* og *keepalive* ville redusere takeovertiden betraktelig. Det ble undersøkt om dette var tilfelle ved å endre parametrene i konfigurasjonsfilen, restarte Heartbeat og gjennomføre testscenariet hvor primærnoden krasjet på nytt. I utgangspunktet var *deadtime* og *keepalive* satt til henholdsvis 500 og 50 millisekunder. Dette vil medføre at man i verste fall får en økning på 550 millisekunder i forhold til den virkelige takeovertiden. Det er derfor ønskelig å holde disse verdiene så lave som mulig. Allikevel er det en begrensning på hvor lavt verdiene kan settes for å ivareta påliteligheten til systemet. Settes *deadtime* for lavt kan man risikere at

det skjer en takeover til tross for at primærnoden ikke har gått ned. Det kan forekomme at pakkene som blir sendt ikke kommer frem til mottaker. Dette indikerer at en node har gått ned, men kan ikke avgjøres med sikkerhet før flere pakker savnes. Man skal derfor vente en viss periode for å sikre seg at noden virkelig har krasjet, og det er denne tidsangivelsen som ligger i parameteren *deadtime*. De beste resultatene som ble oppnådd var med *deadtime* satt til 30 millisekunder og *keepalive* til 10 millisekunder. Et gjennomsnitt av takeovertidene med disse parameterverdiene er vist i figur 8.17, sammen med resultatet fra de to andre testmetodene.

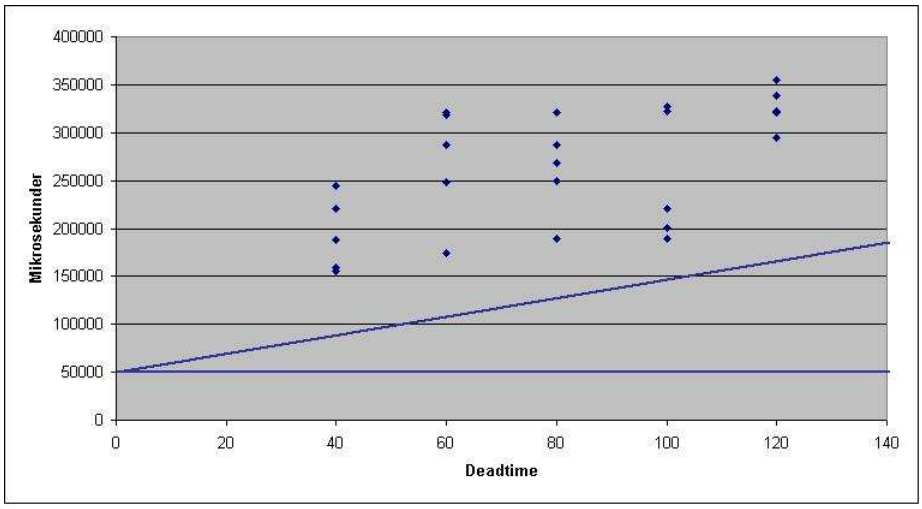


Figur 8.17: Forskjellen i takeovertid mellom de tre metodene etter tuning på Heartbeat parametre

Resultatene fra disse målingene er helt klart mye bedre enn de som opprinnelig ble foretatt. Dette skyldes at parameterverdiene for *deadtime* og *keepalive* opprinnelig var satt til henholdsvis 500 og 50 millisekunder. Det tar da 500 millisekunder før Heartbeat erklærer noden for død, og dette øker naturligvis takeovertiden betraktelig. I tillegg kan det oppstå en ekstra differanse på 50 millisekunder på grunn av *keepalive* verdien, som illustrert i figur 8.14. Dette kan stemme bra med differansen mellom takeovertiden for disse parameterverdiene og de som opprinnelig ble brukt i første testrunde.

Det ville være interessant å undersøke om det eksisterer en sammenheng mellom takeovertiden og Heartbeat parametrene. Det ble fremlagt et forslag om en mulig lineær sammenheng mellom takeovertid og produktet t*n, hvor t er tiden mellom hvert heartbeat som blir sendt (*keepalive*) og n er antall heartbeats som må mistes før en node kan erklæres død (*deadtime*/*keepalive*). Disse resultatene er plottet i et koordinatsystem med takeovertid i mikrosekunder på y-aksen og t*n = *keepalive* * (*deadtime*/*keepalive*) = *deadtime* på x-aksen. Det ble foretatt

flere målinger med ulike *deadtime* og *keepalive* verdier, og resultatet er vist i figur 8.18. *keepalive* verdien er satt til å være 1/4 av *deadtime* slik at forholdet mellom disse parametrene holdes konstant over alle målinger.



Figur 8.18: Sammenheng mellom takeovertid og *deadtime*-parameteren i Heartbeat

Den horisontale linjen som skjærer y-aksen i omtrent 50 000 mikrosekunder representerer prosesseringsoverheaden, det vil si tidsintervallet som alltid vil gå med til takeover uavhengig av deteksjonstid. Prosesseringsoverheaden er estimert ut fra den beste takeovertid-målingen som er foretatt. Målingen ble utført med *deadtime* på 30 millisekunder, og hadde verdien 80 177 mikrosekunder. Da dette er den laveste av alle utførte tidsmålinger, er det rimelig å anta at den nærmer seg laveste oppnåelige tid for takeoverprosedyren. Siden deteksjonstiden på 30 millisekunder ikke skal inngå i prosesseringsoverheaden, subtraheres denne fra den målte verdien på 80 117 mikrosekunder. Dermed blir den estimerte verdien omlag 50 000 mikrosekunder. Dette er kun et overslag, men virker å gi et realistisk bilde på situasjonen.

Den hellende linjen i figur 8.18 representerer en teoretiske kurve for lavest oppnåelige takeovertider. Kurven har en kontinuerlig avstand på *deadtime* fra den horisontale linjen som angir prosesseringsoverhead. Dette er naturlig da *deadtime* bestandig vil angi den teoretiske tidsøkningen for takeovertiden i forhold til prosesseringsoverheaden, nemlig hvor lang periode som kan passere før en node erklæres død. Spredningen av punkter burde teoretisk sett ligge på oversiden av den hellende linjen. For hver verdi av *deadtime* forventes det at punktene har en variasjon i verdi som ligger innen et intervall på t millisekunder over den hellende linjen, det vil si størrelsen på *keepalive*. Årsaken til dette er forklart tidligere, og illustrert i figur 8.14.

Som det fremgår av figur 8.18 ligger målingene foretatt i dette prosjektet en del høyere enn de teoretisk angitte verdiene. Det er allikevel en tendens til at punktene danner en rett linje, dog med noe større spredning enn ønskelig. De

beste takeovermålingene har en verdi på omtrent 50 millisekunder over den teoretisk optimale kurven.

Kapittel 9

Konklusjon og videre arbeid

9.1 Konklusjon

Første skritt i tilnærmingen til problemstillingen var å utvide løsningen som ble utviklet i [4] til å omfatte to noder. Dette ble realisert ved å sette opp en lokal installasjon av Debian Linux på to arbeidsstasjoner, installere og konfigurere Heartbeat og verifisere at den ene noden kunne detektere et krasj hos den andre ved hjelp av Hearbeat API'et. Ved å integrere kallene til Heartbeat API'et i den eksisterende løsningen, var det mulig å kjøre en enkel tilstandsløs applikasjon med takeover på to noder.

Takeover for applikasjoner med en tilstand krever en implementasjon av *Checkpoint Service* (CKPT). CKPT avhenger i henhold til AIS standarden av *Cluster Membership Service* (CLM). Siden implementering av denne delen lå utenfor problemstillingen, er ikke funksjonaliteten bak grensesnittet som CLM definerer realisert. For å kunne håndtere checkpointing var det likevel nødvendig for systemet å kunne administrere medlemskap i clusteret, og det er her Heartbeat kommer inn i bildet. Heartbeat tar seg av deteksjon av node-krasj, kommunikasjon og administrasjon av clusteret, og fungerer dermed som et surrogat for CLM. Heartbeat inkluderer også elementer fra den delen av AIS som kalles *Message Service*. Bruken av Heartbeat medfører at clusterets størrelse begrenses til å omfatte to noder, men for eksperimenteringen i dette prosjektet var ett nodepar tilstrekkelig. Det påpekes at Heartbeat er under stadig utvikling, og et av hovedmålene for det nye Heartbeat designet er å eliminere begrensningen på clusterets størrelse.

Den implementerte løsningen omfatter de basisfunksjonene i *Checkpoint Service* som er nødvendig for at overføring av applikasjonens tilstand fra primær- til standbynode skal fungere tilfredsstillende. Det er foretatt tester der man kjører tre instanser av samme applikasjon på hver node og fremprovoserer krasj på primærnoden. Dette har vist at systemet er i stand til å håndtere takeover for flere applikasjoner parallelt.

Et av målene med standarden er at biblioteket skal kunne tjene en hvilken som helst applikasjon som ønsker å være høytilgjengelig, enten det måtte være et

databasesystem eller en webtjener. Den implementerte løsningen håndterer per i dag skriving av checkpoints med ubegrenset størrelse, men kun en seksjon av gangen. For mer avanserte applikasjoner (for eksempel et databasesystem) vil den informasjonen som skal chekcpointes sannsynligvis være nokså omfattende og det vil dermed være ønskelig å kunne organisere denne informasjonen logisk i flere ulike seksjoner. Det gjenstår dermed noe arbeid før en vilkårlig applikasjon kan benytte det implementerte biblioteket for å oppnå høy tilgjengelighet. Videre er implementasjonen begrenset til kun å omfatte synkron skriving av checkpoints. Asynkron skriving er så langt ikke realisert.

Eksperimenter har indikert at den implementerte løsningen raskt detekterer nodekrasj og utfører den påfølgende takeoverprosedyren. Takeovertiden påvirkes minimalt av størrelsen på checkpointet. Ved tuning på Heartbeat-parametrene var det mulig å oppnå svært gode tider på takeover i forbindelse med nodekrasj. Dette er med på å bygge opp under hypotesen om at Heartbeat er en velfungerende og stabil komponent som oppfyller de krav som et høytilgjengelig system krever. I en feilfri kontekst viser det seg at overhead forbundet med checkpointing kun medfører små forsinkelser. Både tidsforbruk og forbruk av ressurser øker i takt med størrelsen på checkpointet. Likevel er målte takeovertider og latens begge ansett for å være tilstrekkelig lave for et høytilgjengelig system. Dette gir brukeren opplevelsen av en kontinuerlig tilgjengelig tjeneste, selv om et nodekrasj skulle finne sted.

For å kunne opprettholde graden av tilgjengelighet over tid, er det helt avgjørende at det eksisterer en praksis for å reparere en node som har feilet. Uten reparering vil systemet før eller siden, uavhengig av antall standbynoder, risikere å stå igjen med kun én node som kan levere tjenester. Dermed vil systemet ikke lenger tolerere krasjfeil. Systemer som for eksempel databasesystemet ClustRa, tilbyr online reparering av noder. Dette medfører minimal nedetid. Kjennskapen til AIS som er opparbeidet gjennom dette prosjektet, indikerer at det ikke er tilrettelagt for en liknende praksis i AIS. Reparering kan imidlertid gjøres ved at noden tas ned og forlater clusteret. Noden kan så vende tilbake og få tildelt rolle som primær- eller standbynode avhengig av hvilken strategi som er valgt for reintegrering av en node.

9.2 Videre arbeid

Fokus i en videreutvikling av den implementerte løsningen vil ligge på tilrettelegging for at en vilkårlig applikasjon skal kunne benytte biblioteket for å oppnå høy tilgjengelighet. Dette innebærer implementering av flere funksjoner i *Checkpoint Service*, samt en tilpasning av enkelte av de funksjonene som allerede er realisert. Ved å implementere asynkron skriving av checkpoints vil man kunne effektivisere checkpointingen ytterligere. Dette stiller imidlertid store krav til synkronisering av ulike replikater for å garantere konsistens.

For at applikasjonen kun skal trenge å forholde seg til ett bibliotek for høytilgjengelighet, bør Heartbeat-kallene implementeres bak det grensesnittet som *Cluster Membership Service* definerer. Systemer som benytter delte ressurser vil i tillegg være avhengig av kallene i *Lock Service.*

I de foreliggende resultatene alternerer checkpointtidene ved 100 bytes mellom to verdinivåer. Årsaken til variasjonen er ikke kartlagt, og i et videre arbeid bør dette være et moment som tas i betraktning. I den forbindelse kan det være verdt å utforske *patch*en angitt i [23].

Bibliografi

[1] Gray, J. and Siewiorek, D.P., *High Availability Computer Systems*, IEEE Computer, September 1991

[2] Flaviu, C., *Understanding Fault-Tolerant Distributed Systems*, IBM Almaden Research Center, 1990

[3] SA Forum, *Service AvailabilityTM Application Interface Specification*, 2003

[4] Lund, I. and Lønningen, A., *Referanseimplementering av AMF*, November 2004

[5] Hjemmesiden til Service Availability Forum, www.saforum.org

[6] Demacs Associates for SA Forum, *Standards for a Service AvailabilityTM Solution*, 2002

[7] Haddad, I., *Moving toward open platforms*, May 2004

[8] Hjemmesiden til Steeleye med beskrivelse av LifeKeeper, www.steeleye.com/products/linux/

[9] Steeleye Technology Inc., *The Linux Standard for High Availability - LifeKeeper for Linux*, 2001

[10] Hjemmesiden til Beowulf, www.beowulf.org/overview/index.html

[11] White paper hentet fra hjemmesiden til Scyld, www.scyld.com: *The Evolution of Linux Clustering*

[12] Hjemmesiden til Transis, www.cs.huji.ac.il/labs/transis

[13] Dolev, D. and Malki, D., *The Transis Approach to High Availability Cluster Communication,* 1996

[14] Hjemmesiden til Horus, www.cs.cornell.edu/Info/Projects/HORUS/

[15] van Renesse, R., Birman, K. P. and Maffeis, S., *Horus: A Flexible Group Communication System*, 1996

[16] Hjemmesiden til OpenAIS, htt../developer.osdl.org/dev/openais

[17] Hjemmesiden til Linux-HA, www.linux-ha.org

[18] Robertson, A., *Linux-HA Heartbeat System Design,* SeSE Labs

[19] Internettside om høytilgjengelige Linux clustere
htt../www1.ap.dell.com/content/topics/global.aspx/power/en/
ps1q01_linux?c=kr&l=en&s=bsd

[20] Internettside om failover med Heartbeat,
htt../www.geocities.com/latompa/ha/apache_heartbeat.html

[21] Internettside om TCP/IP, www.tcpipguide.com

[22] Internettside om UDP,
htt../compnetworking.about.com/od/networkprotocols/l/aa071200a.htm

[23] Internettside om TCP_NODELAY fix, htt../www.icase.edu/coral/LinuxTCP.html

Tillegg A

Kildekode

amfinternal.h

```
/*******************************************************
 amfinternal.h

 Header for the file which includes internal
 functionality of the AMF framework

 Authors: Anja Lønningen & Ingunn Lund, fall 2004
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *******************************************************/
#include "amfstruct.h"
#include "constants.h"
struct amfAppInstance* amfPool[POOL_SIZE];
SaErrorT amfInitHandlePool();
SaErrorT createAmfHandle(
     struct amfAppInstance *pAmfInstance);
SaErrorT releaseAmfHandle(
     struct amfAppInstance *pAmfInstance);
SaErrorT createAmfInstance(
     struct amfAppInstance *pAmfInstance,
     struct amfCallbacks *amfCallbacks,
     int fd);
SaErrorT deleteAmfInstance(
     struct amfAppInstance *pAmfInstance);
SaErrorT createAmfCompInstance(
     SaAmfHandleT handle,
     SaNameT compName,
     SaNameT proxyCompName);
SaErrorT deleteAmfCompInstance(
     SaAmfHandleT handle,
     SaNameT compName);
```

amfinternal.c

```
/*********************************************************
 amfinternal.c

 The file includes internal functionality of
 the AMF framework

 Authors: Anja Lønningen & Ingunn Lund, fall 2004
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *********************************************************/
#include <stdio.h>
#include "amfinternal.h"
SaErrorT amfInitHandlePool()
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      amfPool[i] = 0;
    }
  return SA_OK;
}
SaErrorT createAmfHandle(
     struct amfAppInstance *pAmfInstance)
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      if(amfPool[i] == 0)
        {
          amfPool[i] = pAmfInstance;
          pAmfInstance->amfHandle = i;
/*        printf("Handle from handle pool: %d\n", i); */
          return SA_OK;
        }
      if(i == POOL_SIZE-1)
        {
          printf("Error in createAmfHandle\n");
          return SA_ERR_BAD_HANDLE;
        }
    }
}
SaErrorT releaseAmfHandle(
     struct amfAppInstance *pAmfInstance)
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      if(i == pAmfInstance->amfHandle)
        {
```

```
          amfPool[i] = 0;
          return SA_OK;
        }
    }
  printf("Error in releaseAmfHandle\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT createAmfInstance(
     struct amfAppInstance *pAmfInstance,
     struct amfCallbacks *amfCallbacks,
     int fd)
{
  struct amfAppInstance amfInstance;
  amfInstance.amfHandle = -1;
  amfInstance.amfCallbacks = *amfCallbacks;
  amfInstance.fd = fd;
  struct amfCompInstance *compInstList = (struct amfCompInstance *)
     malloc(sizeof(struct amfCompInstance)*COMPLIST_SIZE);
  amfInstance.amfCompInstList = compInstList;
  *pAmfInstance = amfInstance;
  if(createAmfHandle(pAmfInstance) == SA_OK)
    {
      return SA_OK;
    }
  printf("Error in createAmfInstance\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT deleteAmfInstance(
     struct amfAppInstance *pAmfInstance)
{
  int i;
  for(i=0; i<POOL_SIZE && pAmfInstance != 0; i++)
    {
      if(releaseAmfHandle(pAmfInstance))
        {
          free(pAmfInstance);
          return SA_OK;
        }
    }
  printf("Error in deleteAmfInstance\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT createAmfCompInstance(
     SaAmfHandleT handle,
     SaNameT compName,
     SaNameT proxyCompName)
{
  int i;
  struct amfAppInstance *pAmfAppInstance;
  struct amfCompInstance amfCompInstance;
```

```
  pAmfAppInstance = amfPool[handle];
  amfCompInstance.compName = compName;
  amfCompInstance.readinessState = SA_AMF_OUT_OF_SERVICE;
  if(proxyCompName.length != 0)
     {
     }
  else
    {
      for(i=0; i<COMPLIST_SIZE; i++)
        {
          if(pAmfAppInstance->amfCompInstList[i].compName.length == 0)
            {
              pAmfAppInstance->amfCompInstList[i] = amfCompInstance;
              break;
            }
          else if(i == COMPLIST_SIZE)
            {
              printf("No availiable slot for new component\n");
              return SA_ERR_BAD_HANDLE;
            }
        }
    }
  return SA_OK;
}
SaErrorT deleteAmfCompInstance(
     SaAmfHandleT handle,
     SaNameT compName)
{
  int i,j;
  int equal = 0;
  struct amfAppInstance *pAmfAppInstance = amfPool[handle];
  struct amfCompInstance amfCompInstance;
  for(i=0; i<COMPLIST_SIZE; i++)
    {
      amfCompInstance = pAmfAppInstance->amfCompInstList[i];
      if(amfCompInstance.compName.length ==
           compName.length)
        {
          equal = 1;
          for(j=0; j<compName.length && equal == 1; j++)
            {
              if(amfCompInstance.compName.value[j] ==
                   compName.value[j])
                {
                  equal = 1;
                  if(j == compName.length-1) break;
                }
              else
                {
                  equal = 0;
```

```
                break;
              }
            }
        }
      if(equal) break;
    }
    if(equal)
      {
        memset(amfCompInstance.compName.value, 0,
             amfCompInstance.compName.length);
        amfCompInstance.compName.length = 0;
        amfCompInstance.readinessState = 0;
      }
    return SA_OK;
}
```

amflib.c

```
/*******************************************************
 amflib.c

 The file includes the implementation of the functions that
 form the interface towards the application. The socket
 connection between the framework (AMF) and the
 application is established in the saAmfInitialize()
 function. The file makes use of the defined Header file of
 SA Forum AIS APIs Version 1.0

 Authors: Anja Lønningen & Ingunn Lund, fall 2004
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *******************************************************/
#include "types.h"
#include "amfstruct.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <sys/un.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#define PORT 9733  /* port, where server waits for connection */
int sockfd;
static const SaAmfCallbacksT *amfCallbackPointers;
/* This is just like the read() system call, */
```

```
/* accept that it will make sure that all */
/* your data goes through the socket. */
int sockRead(int sockfd, char *buf, size_t count)
{
  size_t bytes_read = 0;
  int this_read;
  while (bytes_read < count) {
    do
         this_read = read(sockfd, buf, count - bytes_read);
    while ( (this_read < 0) && (errno == EINTR) );
    if (this_read < 0)
         return this_read;
    else if (this_read == 0)
         return bytes_read;
    bytes_read += this_read;
    buf += this_read;
  }
  return count;
}
SaErrorT saAmfInitialize(
     SaAmfHandleT *amfHandle,
     const SaAmfCallbacksT *amfCallbacks,
     const SaVersionT *version)
{
  int result, len;
  struct sockaddr_in address;
  struct hostent *host;
  char name[] = "localhost";
  struct amfInitData initData;
  int bytes_read;
  struct amfCallbacks amfCallbacksBool;
  int flag = 1;
  SaAmfHandleT handle;
  void *pBuffer;
  struct amfInitData *pData;
  int *pCode;
  /* Create a socket connection to the main procress */
  if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) == -1)
    {
      printf("Error in socket creation\n");
      exit (1);
    }
  host = gethostbyname(name);
  address.sin_family = AF_INET;
  address.sin_addr = *((struct in_addr *) host->h_addr);
  address.sin_port = htons(PORT);
  bzero (&(address.sin_zero), 8);
  result = connect (sockfd,
       (struct sockaddr *) &address, sizeof (address));
  if (result == -1)
```

```
  {
    printf("I'm not able to connect to server\n");
    printf("Errno is %d\n", errno);
    printf("Sleep 10 seconds...");
    sleep(10);
    exit (1);
  }
result = setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY,
     (char *) &flag, sizeof(int));
if (result < 0)
  {
    printf("setsockopt(tcp_nodelay) failed\n");
    fflush(stdout);
    sleep(3);
    exit(1);
  }
amfCallbackPointers = amfCallbacks;
if(amfCallbacks->saAmfHealthcheckCallback != NULL)
    amfCallbacksBool.amfHealthcheckCallback = 1;
if(amfCallbacks->saAmfReadinessStateSetCallback != NULL)
    amfCallbacksBool.amfReadinessStateSetCallback = 1;
if(amfCallbacks->saAmfComponentTerminateCallback != NULL)
    amfCallbacksBool.amfComponentTerminateCallback = 1;
if(amfCallbacks->saAmfCSISetCallback != NULL)
    amfCallbacksBool.amfCSISetCallback = 1;
if(amfCallbacks->saAmfCSIRemoveCallback != NULL)
    amfCallbacksBool.amfCSIRemoveCallback = 1;
if(amfCallbacks->saAmfProtectionGroupTrackCallback != NULL)
    amfCallbacksBool.amfProtectionGroupTrackCallback = 1;
if(amfCallbacks->saAmfExternalComponentRestartCallback != NULL)
    amfCallbacksBool.amfExternalComponentRestartCallback = 1;
if(amfCallbacks->saAmfPendingOperationConfirmCallback != NULL)
    amfCallbacksBool.amfPendingOperationConfirmCallback = 1;
initData.amfCallbacks = amfCallbacksBool;
initData.version = *version;
pBuffer = (void *) malloc(sizeof(int)+
     sizeof(struct amfInitData));
memset(pBuffer, 0, sizeof(int)+
     sizeof(struct amfInitData));
pCode = pBuffer;
pData = pBuffer + sizeof(int);
*pCode = INITIALIZE;
*pData = initData;
len = sizeof(int)+sizeof(struct amfInitData);
if ((result = send (sockfd, pBuffer, len, 0)) == -1)
  {
     printf("message sending not succesful\n");
    exit (1);
  }
free(pBuffer);
```

```
  bytes_read = read(sockfd, &handle, sizeof(int));
  if (bytes_read <= 0)
    {
      /* The other side may have closed unexpectedly */
      return -1;
    }
  *amfHandle = handle;
  return SA_OK;
}
SaErrorT saAmfFinalize(const SaAmfHandleT *amfHandle)
{
  close (sockfd); /* Tror denne skal være med her... */
  return SA_OK;
}
SaErrorT saAmfComponentRegister(
     const SaAmfHandleT *amfHandle,
     const SaNameT *compName,
     const SaNameT *proxyCompName)
{
  int result, len;
  void *pBuffer;
  struct amfCompRegData *pData;
  int *pCode;
  struct amfCompRegData regData;
  regData.amfHandle = *amfHandle;
  regData.compName = *compName;
  regData.proxyCompName = *proxyCompName;
  pBuffer = (void *) malloc(sizeof(int)+
       sizeof(struct amfCompRegData));
  memset(pBuffer, 0, sizeof(int)+
       sizeof(struct amfCompRegData));
  pCode = pBuffer;
  pData = pBuffer + sizeof(int);
  *pCode = COMPREG;
  *pData = regData;
  len = sizeof(int)+sizeof(struct amfCompRegData);
  if ((result = send (sockfd, pBuffer, len, 0)) == -1)
    {
      printf("message sending not succesful\n");
      exit (1);
    }
  free(pBuffer);
  return SA_OK;
}
SaErrorT saAmfComponentUnregister(
      const SaAmfHandleT * amfHandle,
      const SaNameT *compName,
      const SaNameT *proxyCompName)
{
  return SA_OK;
```

```
}
SaErrorT saAmfResponse(
     SaInvocationT invocation,
     SaErrorT error)
{
  int result, len;
  void *pBuffer;
  struct amfResponseData *pData;
  int *pCode;
  struct amfResponseData responseData;
  responseData.invocation = invocation;
  responseData.error = error;
  pBuffer = (void *) malloc(sizeof(int)+
       sizeof(struct amfResponseData));
  memset(pBuffer, 0, sizeof(int)+
       sizeof(struct amfResponseData));
  pCode = pBuffer;
  pData = pBuffer + sizeof(int);
  *pCode = RESPONSE;
  *pData = responseData;
  len = sizeof(int)+sizeof(struct amfResponseData);
  if ((result = send (sockfd, pBuffer, len, 0)) == -1)
    {
      printf("message sending not succesful\n");
      exit (1);
    }
  free(pBuffer);
  return SA_OK;
}
SaErrorT saAmfDispatch(const SaAmfHandleT *amfHandle,
     SaDispatchFlagsT dispatchFlags)
{
  static int blocking = 1;
  int opts;
  void *buffer =  (void *) malloc(500);
  int *pCode = buffer;
  void *pData = buffer + sizeof(int);
  int bytesRead;
  memset(buffer, 0, 500);

  if(dispatchFlags == SA_DISPATCH_ONE)
    {
      /* case AMFREADINESSSTATESETCALLBACK */
      struct amfReadinessStateSetData *amfReadinessStateSetData;
      if(blocking)
        {
          opts = fcntl(sockfd,F_GETFL);
          if (opts < 0)
            {
              perror("fcntl(F_GETFL)");
```

```
            exit(1);
          }
        opts = (opts | O_NONBLOCK);
        if (fcntl(sockfd,F_SETFL,opts) < 0)
          {
            perror("fcntl(F_SETFL)");
            exit(1);
          }
        blocking = 0;
      }
    bytesRead = sockRead(sockfd, buffer, sizeof(int));
/*       if(bytesRead == 0) printf("BytesRead 1 == 0\n"); */
    if(bytesRead == sizeof(int))
      {
        switch(*pCode)
          {
          case AMFREADINESSSTATESETCALLBACK:
            bytesRead = read(sockfd,
                 pData,
                 sizeof(struct amfReadinessStateSetData));
            if(bytesRead == 0) printf("BytesRead == 0\n");
            amfReadinessStateSetData =
            ((struct amfReadinessStateSetData*)pData);
            amfCallbackPointers->saAmfReadinessStateSetCallback(
                 amfReadinessStateSetData->invocation,
                 &(amfReadinessStateSetData->compName),
                 amfReadinessStateSetData->readinessState);
            break;
          default:
            break;
          }
      }
  }
 free(buffer);
 return SA_OK;
}
```

amfstruct.h

```
/*****************************************************
 amfstruct.h
 The file includes the declaration of data structures
 that are used in data transmission between the
 application and AMF.
 Authors: Anja Lønningen & Ingunn Lund, fall 2004
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "ais.h"
struct amfCallbacks
```

```
{
  int amfHealthcheckCallback;
  int amfReadinessStateSetCallback;
  int amfComponentTerminateCallback;
  int amfCSISetCallback;
  int amfCSIRemoveCallback;
  int amfProtectionGroupTrackCallback;
  int amfExternalComponentRestartCallback;
  int amfExternalComponentControlCallback;
  int amfPendingOperationConfirmCallback;
};
struct amfAppInstance
{
  SaAmfHandleT amfHandle;
  struct amfCallbacks amfCallbacks;
  struct amfCompInstance *amfCompInstList;
  int fd;
};
struct amfCompInstance
{
  SaNameT compName;
  SaAmfReadinessStateT readinessState;
};
struct amfInitData
{
  struct amfCallbacks amfCallbacks;
  SaVersionT version;
};
struct amfCompRegData
{
  SaAmfHandleT amfHandle;
  SaNameT compName;
  SaNameT proxyCompName;
};
struct amfResponseData
{
  SaInvocationT invocation;
  SaErrorT error;
};
struct amfReadinessStateSetData
{
  SaInvocationT invocation;
  SaNameT compName;
  SaAmfReadinessStateT readinessState;
};
```

ckptinternal.h

```
/*****************************************************
```

```
ckptinternal.h
Header for the file which includes internal
functionality of the AIS Checkpoint Service.
Authors: Anja Lønningen & Ingunn Lund, spring 2005
Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
****************************************************/
#include "ckptstruct.h"
#include "constants.h"
struct ckptAppInstance* ckptPool[POOL_SIZE];
struct ckptCheckpointReplica* ckptCheckpointPool[POOL_SIZE];
SaErrorT ckptInitHandlePool();
SaErrorT createCkptHandle(
     struct ckptAppInstance *pCkptInstance);
SaErrorT releaseCkptHandle(
     struct ckptAppInstance *pCkptInstance);
SaErrorT createCkptInstance(
     struct ckptAppInstance *pCkptInstance,
     struct ckptCallbacks *ckptCallbacks, int fd);
SaErrorT deleteCkptInstance(
     struct ckptAppInstance *pCkptInstance);
SaErrorT ckptInitCheckpointHandlePool();
SaErrorT createCkptCheckpointHandle(
     struct ckptCheckpointReplica *pCkptCheckpointReplica);
SaErrorT createCkptCheckpointReplica(
     struct ckptCheckpointReplica *pCkptCheckpointReplica,
     SaCkptCheckpointCreationFlagsT flags,
     SaSizeT checkpointSize,
     SaTimeT retentionDuration,
     SaUint32T maxSections,
     SaSizeT maxSectionSize,
     SaUint32T maxSectionIdSize,
     SaCkptCheckpointOpenFlagsT openFlags);
struct ckptAppInstance* findCkptAppInstance(int fd);
struct ckptCheckpointReplica* findCkptReplica(
     SaCkptCheckpointHandleT *checkpointHandle);
struct ckptSection * createCkptSection(
     SaCkptSectionIdT sectionId,
     SaSizeT dataSize,
     SaOffsetT dataOffset);
SaErrorT localCheckpointWrite(
     SaCkptCheckpointHandleT *checkpointHandle,
     SaCkptIOVectorElementT *ioVector,
     SaUint32T *numberOfElements);
SaErrorT checkpointRead(
     SaCkptCheckpointHandleT *checkpointHandle,
     SaCkptIOVectorElementT *ioVector,
     SaUint32T *numberOfElements,
     SaUint32T *erroneousVectorIndex);
SaErrorT remoteCheckpointWrite(
     SaCkptCheckpointHandleT *checkpointHandle,
```

```
     SaCkptIOVectorElementT *ioVector,
     SaUint32T *numberOfElements,
     int * standbysock);
void setBlockingSock(int * sock);
void setNonBlockingSock(int * sock);
```

ckptinternal.c

```
/*****************************************************
 ckptinternal.c
 The file includes internal functionality of the
 AIS Checkpoint service.
 Authors: Anja Lønningen & Ingunn Lund, spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/un.h>
#include <unistd.h>
#include <netdb.h>
#include <errno.h>
#include <fcntl.h>
#include "ckptinternal.h"
#include "types.h"
SaErrorT ckptInitHandlePool()
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      ckptPool[i] = 0;
    }
  return SA_OK;
}
SaErrorT createCkptHandle(
     struct ckptAppInstance *pCkptInstance)
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      if(ckptPool[i] == 0)
        {
          ckptPool[i] = pCkptInstance;
          pCkptInstance->ckptHandle = i;
          return SA_OK;
        }
      if(i == POOL_SIZE-1)
```

```
        {
          printf("Error in createCkptHandle\n");
          return SA_ERR_BAD_HANDLE;
        }
    }
}
SaErrorT releaseCkptHandle(
     struct ckptAppInstance *pCkptInstance)
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      if(i == pCkptInstance->ckptHandle)
        {
          ckptPool[i] = 0;
          return SA_OK;
        }
    }
  printf("Error in releaseCkptHandle\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT createCkptInstance(
     struct ckptAppInstance *pCkptInstance,
     struct ckptCallbacks *ckptCallbacks,
     int fd)
{
  struct ckptAppInstance ckptInstance;
  ckptInstance.ckptHandle = -1;
  ckptInstance.ckptCallbacks = *ckptCallbacks;
  ckptInstance.fd = fd;
  *pCkptInstance = ckptInstance;
  if(createCkptHandle(pCkptInstance) == SA_OK)
    {
      return SA_OK;
    }
  printf("Error in createCkptInstance\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT deleteCkptInstance(
     struct ckptAppInstance *pCkptInstance)
{
  int i;
  for(i=0; i<POOL_SIZE && pCkptInstance != 0; i++)
    {
      if(releaseCkptHandle(pCkptInstance))
        {
          free(pCkptInstance);
          return SA_OK;
        }
    }
```

```
  printf("Error in deleteCkptInstance\n");
  return SA_ERR_BAD_HANDLE;
}
SaErrorT ckptInitCheckpointHandlePool()
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      ckptCheckpointPool[i] = 0;
    }
  return SA_OK;
}
SaErrorT createCkptCheckpointHandle(
     struct ckptCheckpointReplica *pCkptCheckpointReplica)
{
  int i = 0;
  for(i; i<POOL_SIZE; i++)
    {
      if(ckptCheckpointPool[i] == 0)
        {
          ckptCheckpointPool[i] = pCkptCheckpointReplica;
          pCkptCheckpointReplica->ckptCheckpointHandle = i;
          return SA_OK;
        }
      if(i == POOL_SIZE-1)
        {
          printf("Error in createCkptCheckpointHandle\n");
          return SA_ERR_BAD_HANDLE;
        }
    }
}
SaErrorT createCkptCheckpointReplica(
     struct ckptCheckpointReplica *pCkptCheckpointReplica,
     SaCkptCheckpointCreationFlagsT flags,
     SaSizeT checkpointSize,
     SaTimeT retentionDuration,
     SaUint32T maxSections,
     SaSizeT maxSectionSize,
     SaUint32T maxSectionIdSize,
     SaCkptCheckpointOpenFlagsT openFlags)
{
  struct ckptCheckpointReplica ckptReplica;
  struct ckptSection ** sections;
  struct ckptSection *section;
  SaCkptSectionIdT sectionId = SA_CKPT_DEFAULT_SECTION_ID;
  ckptReplica.ckptCheckpointHandle = -1;
  ckptReplica.creationFlags = flags;
  ckptReplica.retentionDuration = retentionDuration;
  ckptReplica.maxSections = maxSections;
  ckptReplica.maxSectionSize = maxSectionSize;
```

```
  ckptReplica.maxSectionIdSize = maxSectionIdSize;
  ckptReplica.openFlags = openFlags;
  sections = (void *) malloc(maxSections*sizeof(struct ckptSection));
  ckptReplica.sections = sections;
  if(maxSections == 1)
    {
      section = createCkptSection(sectionId, 0 /*maxSectionSize*/, 0);
      ckptReplica.sections[0] = section;
    }
  *pCkptCheckpointReplica = ckptReplica;
  if(createCkptCheckpointHandle(pCkptCheckpointReplica) == SA_OK)
    {
      return SA_OK;
    }
  printf("Error in createCkptCheckpointReplica\n");
  return SA_ERR_BAD_HANDLE;
}
struct ckptAppInstance* findCkptAppInstance(int fd)
{
  int i;
  for(i=0;i<POOL_SIZE;i++)
    {
      if(ckptPool[i] != 0)
        {
          if(ckptPool[i]->fd == fd)
            {
              return ckptPool[i];
            }
        }
    }
  return NULL;
}
struct ckptCheckpointReplica* findCkptReplica(
     SaCkptCheckpointHandleT *checkpointHandle)
{
  int i;
  for(i=0;i<POOL_SIZE;i++)
    {
      if(ckptCheckpointPool[i] != 0)
        {
          if(ckptCheckpointPool[i]->ckptCheckpointHandle == *checkpointHandle)
            {
              return ckptCheckpointPool[i];
            }
        }
    }
  return NULL;
}
struct ckptSection * createCkptSection(
     SaCkptSectionIdT sectionId,
```

```
    SaSizeT dataSize,
    SaOffsetT dataOffset)
{
  struct ckptSection * section =
       (struct ckptSection *) malloc(sizeof(struct ckptSection));
  section->sectionId = sectionId;
  section->dataBuffer = NULL;
  section->dataSize = dataSize;
  section->dataOffset = dataOffset;
  return section;
}
SaErrorT localCheckpointWrite(
    SaCkptCheckpointHandleT *checkpointHandle,
    SaCkptIOVectorElementT *ioVector,
    SaUint32T *numberOfElements)
{
  int i, k;
  struct ckptCheckpointReplica * ckptReplica =
       findCkptReplica(checkpointHandle);
  struct ckptSection * section;
  int * pointer;
  void * buffer;
  SaCkptIOVectorElementT ioVectorElement = ioVector[0];
  for(i=0;i<ckptReplica->maxSections;i++)
    {
      if(ckptReplica->sections[i]->sectionId.id ==
           ioVectorElement.sectionId.id && ckptReplica->sections[i]->
           sectionId.idLen == ioVectorElement.sectionId.idLen)
        {
          section = ckptReplica->sections[i];
          break;
        }
    }
  if(section == NULL)
    {
      printf("ckptInternal: Section er NULL\n");
      sleep(1);
      /* Lag ny section med sectionId som er angitt i ioVector */
      /* Sett section til å peke til denne nye sectionen */
    }
  section->dataSize = ioVector[0].dataSize;
  buffer = (void*) malloc(section->dataSize);
  memcpy(buffer, ioVectorElement.dataBuffer,
       ioVectorElement.dataSize);
  if(section->dataSize != 0) free(section->dataBuffer);

  ckptReplica->sections[i]->dataBuffer = buffer;
  return SA_OK;
}
void setBlockingSock(int * sock)
```

```
{
  int opts;
  opts = fcntl(*sock,F_GETFL);
  if (opts < 0)
    {
      perror("fcntl(F_GETFL)");
      exit(1);
    }
  opts = (opts & ~O_NONBLOCK);
  if (fcntl(*sock,F_SETFL,opts) < 0)
    {
      perror("fcntl(F_SETFL)");
      exit(1);
    }
  /* Blocking satt */
}
void setNonBlockingSock(int * sock)
{
  int opts;
  opts = fcntl(*sock,F_GETFL);
  if (opts < 0)
    {
      perror("fcntl(F_GETFL)");
      exit(1);
    }
  opts = (opts | O_NONBLOCK);
  if (fcntl(*sock,F_SETFL,opts) < 0)
    {
      perror("fcntl(F_SETFL)");
      exit(1);
    }
  /* Non-blocking satt */
}
SaErrorT remoteCheckpointWrite(
     SaCkptCheckpointHandleT *checkpointHandle,
     SaCkptIOVectorElementT *ioVector,
     SaUint32T *numberOfElements,
     int * standbysock)
{
  void *pBuffer;
  int bytes_read;
  int len = 0;
  int result;
  SaErrorT res;
  int i,k;
  int code = CKPTREMOTEWRITE;
  void *bufferStart;
  int *pointer;
  len += sizeof(int); /* code */
  len += sizeof(int); /* len */
```

```
len += sizeof(SaCkptCheckpointHandleT); /* checkpointHandle */
len += sizeof(SaUint32T); /* numberOfElements */
for(i=0;i<*numberOfElements;i++)
  {
    len += sizeof(SaUint32T); /* idLen */
    len += ioVector[i].sectionId.idLen; /* id */
    len += sizeof(SaSizeT); /* dataSize */
    len += ioVector[i].dataSize; /* dataBuffer */
    len += sizeof(SaOffsetT); /* dataOffset */
    len += sizeof(SaSizeT); /* readSize */
  }
pBuffer = (void *) malloc(len);
memset(pBuffer, 0, len);
bufferStart = pBuffer;
pointer = bufferStart;
memcpy(pBuffer, &code, sizeof(int));
pBuffer += sizeof(int);
memcpy(pBuffer, &len, sizeof(int));
pBuffer += sizeof(int);

memcpy(pBuffer, checkpointHandle,
     sizeof(SaCkptCheckpointHandleT));
pBuffer += sizeof(SaCkptCheckpointHandleT);
memcpy(pBuffer, numberOfElements, sizeof(SaUint32T));
pBuffer += sizeof(SaUint32T);
for(i=0;i<*numberOfElements;i++)
  {
    memcpy(pBuffer, &ioVector[i].sectionId.idLen,
         sizeof(SaUint32T));
    pBuffer += sizeof(SaUint32T);

    if(ioVector[i].sectionId.idLen != 0)
      {
        memcpy(pBuffer, ioVector[i].sectionId.id,
             ioVector[i].sectionId.idLen);
        pBuffer += ioVector[i].sectionId.idLen;
      }
    memcpy(pBuffer, &ioVector[i].dataSize, sizeof(SaSizeT));
    pBuffer += sizeof(SaSizeT);
    memcpy(pBuffer, ioVector[i].dataBuffer, ioVector[i].dataSize);
    pBuffer += ioVector[i].dataSize;
    memcpy(pBuffer, &ioVector[i].dataOffset, sizeof(SaOffsetT));
    pBuffer += sizeof(SaOffsetT);
    memcpy(pBuffer, &ioVector[i].readSize, sizeof(SaSizeT));
    pBuffer += sizeof(SaSizeT);
  }
if ((result = send (*standbysock, bufferStart, len, 0)) == -1)
  {
    printf("CkptInternal: Sending remote writeData not succesful\n");
    sleep(3);
```

```
      exit (1);
    }
  free(bufferStart);
  /* Set blocking */
  setBlockingSock(standbysock);
  bytes_read = read(*standbysock, &res, sizeof(SaErrorT));
  if (bytes_read <= 0)
    {
      printf("The other side may have closed unexpectedly\n");
      /* The other side may have closed unexpectedly */
      return -1;
    }
  if(res != SA_OK)
    {
      printf("Checkpoint write failed\n");
      exit(1);
    }
  /* Set non-blocking */
  setNonBlockingSock(standbysock);
  return SA_OK;
}
SaErrorT checkpointRead(
     SaCkptCheckpointHandleT *checkpointHandle,
     SaCkptIOVectorElementT *ioVector,
     SaUint32T *numberOfElements,
     SaUint32T *erroneousVectorIndex)
{
  int i, j;
  struct ckptCheckpointReplica * ckptReplica =
       findCkptReplica(checkpointHandle);
  struct ckptSection * section;
  int * pointer;

  *erroneousVectorIndex = -1;
  if(ckptReplica->sections == NULL)
    {
      *erroneousVectorIndex = 0;
      return SA_OK;
    }
  for(j=0;j<*numberOfElements;j++)
    {
      for(i=0;i<ckptReplica->maxSections;i++)
        {
          if(ckptReplica->sections[i]->sectionId.id ==
             ioVector[j].sectionId.id && ckptReplica->sections[i]->
             sectionId.idLen == ioVector[j].sectionId.idLen)
            {
              section = ckptReplica->sections[i];;
            }
        }
```

```
      if(section == NULL)
        {
          printf("ckptInternal: Sectionid does not exist\n");
          *erroneousVectorIndex = 0;
          return SA_OK;
        }

      if(section->dataBuffer == NULL)
        {
          *erroneousVectorIndex = 0;
          return SA_OK;
        }

      ioVector[j].dataBuffer = section->dataBuffer;
      ioVector[j].dataSize = section->dataSize;
      ioVector[j].readSize = section->dataSize;
    }
  return SA_OK;
}
```

ckptlib.c

```
/*****************************************************
 ckptlib.c
 The file includes the implementation of the functions
 that form the interface towards the application. The
 file makes use of the defined header file of SA Forum
 AIS APIs Version 1.0
 Authors: Anja Lønningen & Ingunn Lund, spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "types.h"
#include "ckptstruct.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/un.h>
#include <unistd.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
extern int sockfd;
static const SaCkptCallbacksT *ckptCallbackPointers;
SaErrorT SaCkptInitialize(
     SaCkptHandleT *ckptHandle,
```

```
    const SaCkptCallbacksT *callbacks,
    const SaVersionT *version)
{
  struct ckptInitData initData;
  int bytes_read;
  struct ckptCallbacks ckptCallbacksBool;
  SaCkptHandleT handle;
  void *pBuffer;
  struct ckptInitData *pData;
  int *pCode;
  int len;
  int result;
  ckptCallbackPointers = callbacks;
  if(callbacks->saCkptCheckpointOpenCallback != NULL)
      ckptCallbacksBool.ckptCheckpointOpenCallback = 1;
  if(callbacks->saCkptCheckpointSynchronizeCallback != NULL)
      ckptCallbacksBool.ckptCheckpointSynchronizeCallback = 1;

  initData.ckptCallbacks = ckptCallbacksBool;
  initData.version = *version;
  pBuffer = (void *) malloc(sizeof(int)+sizeof(struct ckptInitData));
  memset(pBuffer, 0, sizeof(int)+sizeof(struct ckptInitData));
  pCode = pBuffer;
  pData = pBuffer + sizeof(int);
  *pCode = CKPTINITIALIZE;
  *pData = initData;
  len = sizeof(int)+sizeof(struct ckptInitData);
  if ((result = send (sockfd, pBuffer, len, 0)) == -1)
    {
      printf("message sending not succesful\n");
      exit (1);
    }
  free(pBuffer);
  bytes_read = read(sockfd, &handle, sizeof(int));
  if (bytes_read <= 0)
    {
      /* The other side may have closed unexpectedly */
      return -1; /* Is this effective on other platforms than linux? */
    }
  *ckptHandle = handle;
  return SA_OK;
}
SaErrorT saCkptCheckpointOpen(
    const SaNameT *ckeckpointName,
    const SaCkptCheckpointCreationAttributesT *checkpointCreationAttributes,
    SaCkptCheckpointOpenFlagsT checkpointOpenFlags,
    SaTimeT timeout,
    SaCkptCheckpointHandleT *checkpointHandle)
{
  SaCkptCheckpointHandleT ckptCheckpointHandle;
```

```
  void *pBuffer;
  struct ckptCheckpointOpenData *pData;
  int *pCode;
  struct ckptCheckpointOpenData openData;
  int bytes_read;
  int len;
  int result;
  pBuffer = (void *) malloc(sizeof(int)+
       sizeof(struct ckptCheckpointOpenData));
  memset(pBuffer, 0, sizeof(int)+
       sizeof(struct ckptCheckpointOpenData));
  pCode = pBuffer;
  pData = pBuffer + sizeof(int);
  openData.ckeckpointName = *ckeckpointName;
  openData.checkpointCreationAttributes = *checkpointCreationAttributes;
  openData.checkpointOpenFlags = checkpointOpenFlags;
  openData.timeout = timeout;
  *pCode = CKPTOPEN;
  *pData = openData;
  len = sizeof(int)+sizeof(struct ckptCheckpointOpenData);
  if ((result = send (sockfd, pBuffer, len, 0)) == -1)
    {
      printf("message sending not succesful\n");
      exit (1);
    }
  free(pBuffer);
  bytes_read = read(sockfd, &ckptCheckpointHandle, sizeof(int));
  if (bytes_read <= 0)
    {
      printf("The other side may have closed unexpectedly\n");
      sleep(2);
      /* The other side may have closed unexpectedly */
      return -1;
    }
  *checkpointHandle = ckptCheckpointHandle;
  return SA_OK;

}
SaErrorT saCkptCheckpointClose(
     const SaCkptCheckpointHandleT *checkpointHandle)
{
  return SA_OK;
}
void setBlocking()
{
  int opts;
  opts = fcntl(sockfd,F_GETFL);
  if (opts < 0)
    {
      perror("fcntl(F_GETFL)");
```

```
      exit(1);
    }
  opts = (opts & ~O_NONBLOCK);
  if (fcntl(sockfd,F_SETFL,opts) < 0)
    {
      perror("fcntl(F_SETFL)");
      exit(1);
    }
  /* Blocking satt */
}
void setNonBlocking()
{
  int opts;
  opts = fcntl(sockfd,F_GETFL);
  if (opts < 0)
    {
      perror("fcntl(F_GETFL)");
      exit(1);
    }
  opts = (opts | O_NONBLOCK);
  if (fcntl(sockfd,F_SETFL,opts) < 0)
    {
      perror("fcntl(F_SETFL)");
      exit(1);
    }
  /* Non-blocking satt */
}
SaErrorT saCkptCheckpointWrite(
     const SaCkptCheckpointHandleT *checkpointHandle,
     const SaCkptIOVectorElementT *ioVector,
     SaUint32T numberOfElements,
     SaUint32T *erroneousVectorIndex)
{
  void *pBuffer;
  int bytes_read;
  int len = 0;
  int result;
  SaErrorT res;
  int i,k;
  int code = CKPTWRITE;
  void *bufferStart;
  int nullpointer;
  len += sizeof(int); /* code */
  len += sizeof(int); /* len */
  len += sizeof(SaCkptCheckpointHandleT); /* checkpointHandle */
  len += sizeof(SaUint32T); /* numberOfElements */
  for(i=0;i<numberOfElements;i++)
    {
      len += sizeof(SaUint32T); /* idLen */
      len += ioVector[i].sectionId.idLen; /* id */
```

```
    len += sizeof(SaSizeT); /* dataSize */
    len += ioVector[i].dataSize; /* dataBuffer */
    len += sizeof(SaOffsetT); /* dataOffset */
    len += sizeof(SaSizeT); /* readSize */
  }
pBuffer = (void *) malloc(len);
memset(pBuffer, 0, len);
bufferStart = pBuffer;
memcpy(pBuffer, &code, sizeof(int));
pBuffer += sizeof(int);
memcpy(pBuffer, &len, sizeof(int));
pBuffer += sizeof(int);

memcpy(pBuffer, checkpointHandle,
     sizeof(SaCkptCheckpointHandleT));
pBuffer += sizeof(SaCkptCheckpointHandleT);
memcpy(pBuffer, &numberOfElements,
     sizeof(SaUint32T));
pBuffer += sizeof(SaUint32T);
for(i=0;i<numberOfElements;i++)
  {
    memcpy(pBuffer, &ioVector[i].sectionId.idLen,
         sizeof(SaUint32T));
    pBuffer += sizeof(SaUint32T);

    if(ioVector[i].sectionId.idLen != 0)
      {
        memcpy(pBuffer, ioVector[i].sectionId.id,
             ioVector[i].sectionId.idLen);
        pBuffer += ioVector[i].sectionId.idLen;
      }
    memcpy(pBuffer, &ioVector[i].dataSize, sizeof(SaSizeT));
    pBuffer += sizeof(SaSizeT);
    memcpy(pBuffer, ioVector[i].dataBuffer,
         (int)(ioVector[i].dataSize));
    pBuffer += ioVector[i].dataSize;
    memcpy(pBuffer, &ioVector[i].dataOffset,
         sizeof(SaOffsetT));
    pBuffer += sizeof(SaOffsetT);
    memcpy(pBuffer, &ioVector[i].readSize,
         sizeof(SaSizeT));
    pBuffer += sizeof(SaSizeT);
  }
if ((result = send (sockfd, bufferStart, len, 0)) == -1)
  {
    printf("Sending writeData not succesful\n");
    sleep(3);
    exit (1);
  }
free(bufferStart);
```

```
  /* Set blocking */
  setBlocking();
  bytes_read = read(sockfd, &res, sizeof(SaErrorT));
  if (bytes_read <= 0)
    {
      printf("The other side may have closed unexpectedly\n");
      /* The other side may have closed unexpectedly */
      return -1;
    }
  if(res != SA_OK)
    {
      printf("Checkpoint write failed\n");
      exit(1);
    }
  /* Set non-blocking */
  setNonBlocking();
  return SA_OK;
}
SaErrorT saCkptCheckpointRead(
     const SaCkptCheckpointHandleT *checkpointHandle,
     SaCkptIOVectorElementT *ioVector,
     SaUint32T numberOfElements,
     SaUint32T *erroneousVectorIndex)
{
  /* Send data */
  void *pBuffer;
  int code = CKPTREAD;
  void *bufferStart;
  int bytes_read;
  int len = 0;
  int result;
  int i;
  /* Recieve data */
  void * readDataBuffer;
  int bufferLen;
  SaCkptSectionIdT sectionId;
  SaUint32T * pSaUint32T;
  SaSizeT * pSaSizeT;
  int * dataBuffer;
  len += sizeof(int); /* code */
  len += sizeof(int); /* len */
  len += sizeof(SaCkptCheckpointHandleT); /* checkpointHandle */
  len += sizeof(SaUint32T); /* numberOfElements */
  len += sizeof(SaCkptSectionIdT)*numberOfElements; /* SectionId's */
  len += sizeof(SaOffsetT)*numberOfElements; /* Offset values */
  /*   len += sizeof(SaUint32T); ../* erroneousVectorIndex */

  pBuffer = (void *) malloc(len);
  memset(pBuffer, 0, len);
  bufferStart = pBuffer;
```

```
 memcpy(pBuffer, &code, sizeof(int));
 pBuffer += sizeof(int);
 memcpy(pBuffer, &len, sizeof(int));
 pBuffer += sizeof(int);

 memcpy(pBuffer, checkpointHandle,
      sizeof(SaCkptCheckpointHandleT));
 pBuffer += sizeof(SaCkptCheckpointHandleT);
 memcpy(pBuffer, &numberOfElements,
      sizeof(SaUint32T));
 pBuffer += sizeof(SaUint32T);
 for(i=0;i<numberOfElements;i++)
   {
     memcpy(pBuffer, &ioVector[i].sectionId.idLen,
          sizeof(SaUint32T));
     pBuffer += sizeof(SaUint32T);

     if(ioVector[i].sectionId.idLen != 0)
       {
         memcpy(pBuffer, ioVector[i].sectionId.id,
              ioVector[i].sectionId.idLen);
         pBuffer += ioVector[i].sectionId.idLen;
       }
     memcpy(pBuffer, &ioVector[i].dataOffset, sizeof(SaOffsetT));
     pBuffer += sizeof(SaOffsetT);
   }
if ((result = send (sockfd, bufferStart, len, 0)) == -1)
   {
     printf("Sending readData not succesful\n");
     sleep(3);
     exit (1);
   }
 free(bufferStart);
 /* Set blocking */
 setBlocking();
 /* Read response from server */
 bytes_read = read(sockfd, &bufferLen, sizeof(int));
 /* Set non-blocking */
 setNonBlocking();

 readDataBuffer = (void *) malloc(bufferLen-(sizeof(int)));
 memset(readDataBuffer, 0, bufferLen-(sizeof(int)));

 bytes_read = sockRead(sockfd, readDataBuffer, bufferLen-(sizeof(int)));
 if(bytes_read == 0) {
   printf("Nr. 6: BytesRead == 0\n");
   fflush(stdout);
 }

 for(i=0;i<(numberOfElements);i++)
```

```
    {
      pSaUint32T = readDataBuffer;
      sectionId.idLen = *pSaUint32T;
      readDataBuffer += sizeof(SaUint32T);

      if(sectionId.idLen == 0)
        {
          sectionId.id = NULL;
                }
      else
        {
          sectionId.id = readDataBuffer;
        }

      readDataBuffer += sectionId.idLen;
      ioVector[i].sectionId = sectionId;

      pSaSizeT = readDataBuffer;
      ioVector[i].dataSize = *pSaSizeT;
      readDataBuffer += sizeof(SaSizeT);

      ioVector[i].dataBuffer = readDataBuffer;
      readDataBuffer += ioVector[i].dataSize;

      pSaSizeT = readDataBuffer;
      ioVector[i].readSize = *pSaSizeT;
      readDataBuffer += sizeof(SaSizeT);
      pSaUint32T = readDataBuffer;
      *erroneousVectorIndex = *pSaUint32T;
      readDataBuffer += sizeof(SaUint32T);
    }
  return SA_OK;
}
```

ckptstruct.h

```
/*****************************************************
 ckptstruct.h
 The file includes the declaration of data structures
 that are used in data transmission between the
 application and the checkpoint service.
 Authors: Anja Lønningen & Ingunn Lund, spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "ais.h"
struct ckptCallbacks
{
  int ckptCheckpointOpenCallback;
  int ckptCheckpointSynchronizeCallback;
```

```
};
struct ckptSection
{
  SaCkptSectionIdT sectionId;
  void * dataBuffer;
  SaSizeT dataSize;
  SaOffsetT dataOffset;
};
struct ckptCheckpointReplica
{
  SaCkptCheckpointHandleT ckptCheckpointHandle;
  struct ckptSection ** sections;
  SaCkptCheckpointCreationFlagsT creationFlags;
  SaTimeT retentionDuration;
  SaUint32T maxSections;
  SaSizeT maxSectionSize;
  SaUint32T maxSectionIdSize;
  SaCkptCheckpointOpenFlagsT openFlags;
};
struct ckptAppInstance
{
  SaCkptHandleT ckptHandle;
  struct ckptCallbacks ckptCallbacks;
  int fd;
  struct ckptCheckpointReplica ckptReplica;
};
struct ckptInitData
{
  struct ckptCallbacks ckptCallbacks;
  SaVersionT version;
};
struct ckptCheckpointOpenData
{
  SaNameT ckeckpointName;
  SaCkptCheckpointCreationAttributesT checkpointCreationAttributes;
  SaCkptCheckpointOpenFlagsT checkpointOpenFlags;
  SaTimeT timeout;
};
struct udpdata
{
  int nodeid;
  int sequencenumber;
};
```

constants.h

```
/*****************************************************
constants.h
The file includes constants that are global to AMF and CKPT
```

```
Authors: Anja Lønningen & Ingunn Lund, fall 2004 / spring 2005
Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
*****************************************************/
#define POOL_SIZE 10
#define CONFIG_FILE "/home/ingunn/Diplom/Kildekode/config"
#define RESULTS_FILE "/home/ingunn/Diplom/Kildekode/results"
#define USAGE_FILE "/home/ingunn/Diplom/Kildekode/usage"
#define COMPLIST_SIZE 5
#define PRIMARYNODE "stud2104"
#define STANDBYNODE "stud2103"
```

counterapp.c

```
/*****************************************************
 counterapp.c
The file forms a user application which for testing
 purposes makes use of the AMF and CKPT.
Authors: Anja Lønningen & Ingunn Lund, spring 2005
Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "ais.h"
#include <stdio.h>
#include <ctype.h>
#include <sys/select.h>
extern int sockfd;
SaAmfReadinessStateT compReadinessState = SA_AMF_OUT_OF_SERVICE;
int number = 0;
SaCkptIOVectorElementT* createIOVector(int* data, int numberOfElements)
{
  int i;
  SaCkptIOVectorElementT element;
  SaCkptIOVectorElementT* ioVector = (SaCkptIOVectorElementT*)
       malloc(numberOfElements*sizeof(SaCkptIOVectorElementT));
  SaCkptSectionIdT sectionId = SA_CKPT_DEFAULT_SECTION_ID;
  for(i=0; i<numberOfElements; i++)
    {
      element.sectionId = sectionId;
      element.dataBuffer = &(data[i]);
      element.dataSize = sizeof(int);
      element.dataOffset = i;
      element.readSize = sizeof(int);
      ioVector[i] = element;
    }
  return ioVector;
}
void printNumber(SaAmfHandleT * amfHandle,
                              SaCkptCheckpointHandleT *checkpointHandle)
{
  int numberOfElements = 1;
```

```
  SaUint32T *erroneousVectorIndex;
  SaCkptIOVectorElementT* ioVector;
  int * pointer;
  while(1)
    {
      if(compReadinessState == SA_AMF_IN_SERVICE)
        {
          printf ("\nNumber: %d\n", number);
          /*Checkpoint*/
          if(saCkptCheckpointWrite(checkpointHandle,
             createIOVector(&number, numberOfElements),
             numberOfElements,
             erroneousVectorIndex) != SA_OK)
            {
              printf("Error while trying to call SaCkptCheckpointWrite\n");
              sleep(3);
              exit(1);
            }
          printf("Number %d checkpointed\n", number);
          number += 1;
        }
      sleep(1);
      saAmfDispatch(amfHandle, SA_DISPATCH_ONE);
    }
}
void amfReadinessStateSetCallback(SaInvocationT invocation,
     const SaNameT *compName,
     SaAmfReadinessStateT readinessState)
{
  compReadinessState = readinessState;
  saAmfResponse(invocation, SA_OK);
}
int main (int argc, char* argv[])
{
  SaAmfHandleT amfHandle;
  SaAmfCallbacksT amfCallbacks;
  const SaVersionT version;
  SaCkptHandleT ckptHandle;
  SaCkptCallbacksT ckptCallbacks;
  char* string;
  fd_set readfd;
  int rc;
  SaNameT counterCompName, proxyCompName;
  SaNameT ckeckpointName;
  SaCkptCheckpointCreationAttributesT checkpointCreationAttributes;
  SaCkptCheckpointOpenFlagsT checkpointOpenFlags =
       SA_CKPT_CHECKPOINT_WRITE;
  SaTimeT timeout = 100000;
  SaCkptCheckpointHandleT checkpointHandle;
  SaUint32T *erroneousVectorIndex;
```

```
int numberOfElements = 1;
SaCkptIOVectorElementT* ioVector;
int * pointer;
strcpy(counterCompName.value, "Number printer");
counterCompName.length =
     strlen(counterCompName.value) + 1;
amfCallbacks.saAmfReadinessStateSetCallback =
     amfReadinessStateSetCallback;

if(saAmfInitialize(&amfHandle, &amfCallbacks,
     &version) != SA_OK)
  {
    printf("Error while trying to call SaAmfInitialize\n");
    exit(1);
  }
if(SaCkptInitialize(&ckptHandle, &ckptCallbacks,
     &version) != SA_OK)
  {
    printf("Error while trying to call SaCkptInitialize\n");
    exit(1);
  }
strcpy(ckeckpointName.value, "Number printer checkpoint");
ckeckpointName.length = strlen(ckeckpointName.value) + 1;
checkpointCreationAttributes.creationFlags =
     SA_CKPT_WR_ALL_REPLICAS;
checkpointCreationAttributes.checkpointSize = sizeof(int);
checkpointCreationAttributes.retentionDuration = 20;
checkpointCreationAttributes.maxSections = 1;
checkpointCreationAttributes.maxSectionSize = 2*sizeof(int);
checkpointCreationAttributes.maxSectionIdSize = sizeof(int);
if(saCkptCheckpointOpen(&ckeckpointName,
   &checkpointCreationAttributes,
   checkpointOpenFlags,
   timeout, &checkpointHandle) != SA_OK)
  {
    printf("Error while trying to call SaCkptCheckpointOpen\n");
    sleep(3);
    exit(1);
  }
if(saCkptCheckpointWrite(&checkpointHandle,
   createIOVector(&number, numberOfElements),
   numberOfElements, erroneousVectorIndex) != SA_OK)
  {
    printf("Error while trying to call SaCkptCheckpointWrite\n");
    sleep(3);
    exit(1);
  }
printf("Initial value of number checkpointed: %d\n", number);

if(saAmfComponentRegister(&amfHandle,
```

```
     &counterCompName, &proxyCompName) != SA_OK)
    {
      printf("Error while trying to call SaAmfComponentRegister\n");
      exit(1);
    }
  while(compReadinessState == SA_AMF_OUT_OF_SERVICE)
    {
      saAmfDispatch(&amfHandle, SA_DISPATCH_ONE);
    }
  if(compReadinessState == SA_AMF_IN_SERVICE)
    {
      if(saCkptCheckpointRead(&checkpointHandle, ioVector =
         createIOVector(NULL, numberOfElements), numberOfElements,
         erroneousVectorIndex) != SA_OK)
        {
          printf("Error while trying to call SaCkptCheckpointRead\n");
          sleep(3);
          exit(1);
        }

      pointer = ioVector[0].dataBuffer;
      printf("Number read from checkpoint: %d\n", *pointer);
      number = *pointer;
    }
  printNumber(&amfHandle, &checkpointHandle);
  return 0;
}
```

main.c

```
/*********************************************************
 main.c
 The file includes the executable main function which
 represents the main process which handles the initialisation
 of the user applications before starting to deal with:
 - new connections
 - incoming data
 - callbacks

 Authors: Anja Lønningen & Ingunn Lund,
 fall 2004 / spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *********************************************************/
#include "sockcom.h"
#include "constants.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <stdarg.h>
#include <heartbeat.h>
#include <hb_api.h>
ll_cluster_t* hb;
int main()
{
  int bufferSize = 100;
  char** appPaths;
  char* buffer;
  char* pPath;
  int i,j;
  int pid;
  int t;
  int status;
  FILE *in;
  /*Heartbeat stuff*/
  unsigned fmask;

  (void)_heartbeat_h_Id;
  (void)_ha_msg_h_Id;
  printf("Main process running with pid: %d\n", getpid());
  /*Starting heartbeat*/
  hb = ll_cluster_new("heartbeat");
  if(hb == NULL) printf("Feil ved oppretting av cluster\n");

  if (hb->llc_ops->signon(hb, "ais")!= HA_OK)
    {
      fprintf(stderr, "Cannot sign on with heartbeat\n");
      fprintf(stderr, "REASON: %s\n", hb->llc_ops->errmsg(hb));
      sleep(5);
      exit(1);
    }
/*   else fprintf(stdout, "Signed on sucessfully\n"); */
  appPaths = (void *) malloc(POOL_SIZE/2);
  for(i=0;i<POOL_SIZE/2;i++)
  {
        appPaths[i] = 0;
  }
  if ((in = fopen(CONFIG_FILE, "r")) == NULL)
    {
      printf("Unable to open the inputfile\n");
      sleep(5);
      exit(1);
```

```
    }
  buffer = (char *) malloc(bufferSize);
  memset(buffer, 0, bufferSize);
  fscanf(in, "%s", buffer);
  for(i=0; !feof(in) && i<POOL_SIZE/2; i++)
    {
      pPath = (char *) malloc(strlen(buffer));
      for(j=0;j<strlen(buffer);j++)
        {
          *(pPath+j) = *(buffer+j);
        }
      appPaths[i] = pPath;
      memset(buffer, 0, bufferSize);
      fscanf(in, "%s", buffer);
    }
  fclose(in);
  for(i=0; i<1 /*i<POOL_SIZE/2 && appPaths[i] != 0*/; i++)
    {
/*       printf("appPaths[%d]: %s\n", i, appPaths[i]); */
      pid = fork();
      if(pid < 0)
        {
          printf("Error in fork()");
          exit(1);
        }
      if(pid == 0) /* Child */
        {
          execl("/usr/X11R6/bin/xterm", "xterm", "-e", appPaths[i], 0);
        }
    }
  sockCommunicate();
  if (hb->llc_ops->signoff(hb) != HA_OK)
    {
      fprintf(stderr, "Cannot sign off from heartbeat.\n");
      fprintf(stderr, "REASON: %s\n", hb->llc_ops->errmsg(hb));
      exit(10);
    }
  if (hb->llc_ops->delete(hb) != HA_OK)
    {
      fprintf(stderr, "REASON: %s\n", hb->llc_ops->errmsg(hb));
      fprintf(stderr, "Cannot delete API object.\n");
      fprintf(stderr, "REASON: %s\n", hb->llc_ops->errmsg(hb));
      exit(11);
    }
  return 0;
}
```

sockcom.h

```
/*****************************************************
 sockcom.h
 Header for the file which includes the main process part of
 the socket communication.
 Authors: Anja Lønningen & Ingunn Lund,
 fall 2004 / spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <errno.h>
void setNonBlocking(int sock);
void buildSelectList();
void handleNewConnection();
void readSocks();
int sockWrite(int sockfd, char *buf, size_t count);
int sockRead(int sockfd, char *buf, size_t count);
int openClusterSocket();
int connectClusterSocket();
int sockHandleFuncCall(int sockfd);
int sockCommunicate();
```

sockcom.c

```
/*****************************************************
 sockcom.c

 The file includes the main process part of the socket
 communication.
 Authors: Anja Lønningen & Ingunn Lund,
 fall 2004 / spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "types.h"
#include "sockcom.h"
#include "amfinternal.h"
#include "ckptinternal.h"
#include <ctype.h>
#include <sys/time.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <heartbeat.h>
#include <hb_api.h>
#include <sys/resource.h>
/* The socket file descriptor for our "listening" socket */
```

```
int sock;
/* Array of connected sockets so we know who we are talking to */
int connectlist[6];
/* Socket file descriptors we want to wake up for, using select() */
fd_set socks;
/* Highest #'d file descriptor, needed for select() */
int highsock;
int primary = 0;
int clustersock = 0;
int standbysock = 0;
extern ll_cluster_t* hb;
void setNonBlocking(int sock)
{
  int opts;
  opts = fcntl(sock,F_GETFL);
  if (opts < 0)
    {
      perror("fcntl(F_GETFL)");
      exit(EXIT_FAILURE);
    }
  opts = (opts | O_NONBLOCK);
  if (fcntl(sock,F_SETFL,opts) < 0)
    {
      perror("fcntl(F_SETFL)");
      exit(EXIT_FAILURE);
    }
  return;
}
void buildSelectList()
{
  /* Current item in connectlist for for loops */
  int listnum;
  /* First put together fd_set for select(), which will
     consist of the sock veriable in case a new connection
     is coming in, plus all the sockets we have already
     accepted. */
  /* FD_ZERO() clears out the fd_set called socks, so that
     it doesn't contain any file descriptors. */
  FD_ZERO(&socks);
  /* FD_SET() adds the file descriptor "sock" to the fd_set,
     so that select() will return if a connection comes in
     on that socket (which means you have to do accept(), etc. */
  FD_SET(sock,&socks);
  if(clustersock != 0) FD_SET(clustersock,&socks);
  /* Loops through all the possible connections and adds
     those sockets to the fd_set */
  for (listnum = 0; listnum < 6; listnum++)
    {
      if (connectlist[listnum] != 0)
        {
```

```
          FD_SET(connectlist[listnum],&socks);
          if (connectlist[listnum] > highsock)
            highsock = connectlist[listnum];
        }
    }
  if(clustersock > highsock) highsock = clustersock;
}
/* Set readinessState for all registered applications */
void setReadinessState(SaAmfReadinessStateT readinessState)
{
  struct amfReadinessStateSetData amfReadinessStateSetData;
  void * outBuffer;
  int * pCode;
  struct amfReadinessStateSetData * pAmfReadinessStateSetData;
  int i, len, result;
  amfReadinessStateSetData.invocation = 1;
  strcpy(amfReadinessStateSetData.compName.value, "");
  amfReadinessStateSetData.compName.length = 0;
  amfReadinessStateSetData.readinessState = readinessState;

  outBuffer = malloc(sizeof(int)+sizeof(struct amfReadinessStateSetData));
  memset(outBuffer, 0, sizeof(int)+sizeof(struct amfReadinessStateSetData));

  pCode = outBuffer;
  pAmfReadinessStateSetData = outBuffer + sizeof(int);

  *pCode = AMFREADINESSSTATESETCALLBACK;
  *pAmfReadinessStateSetData = amfReadinessStateSetData;

  len = sizeof(int)+sizeof(struct amfReadinessStateSetData);

  for(i=0; i<POOL_SIZE; i++)
    {
      if(amfPool[i] != 0)
        {
          if ((result = sockWrite (amfPool[i]->fd, outBuffer, len)) == -1)
            {
              printf("Callback sending not succesful\n");
              exit (1);
            }
        }
    }
  if(readinessState == SA_AMF_IN_SERVICE)
    primary = 1;
  else if (readinessState == SA_AMF_OUT_OF_SERVICE)
    primary = 0;
  free(outBuffer);
}
void handleNewConnection(int * socket)
{
```

```
  /* Current item in connectlist for for loops */
  int listnum;
  /* Socket file descriptor for incoming connections */
  int connection;
  int flag = 1;
  int result;
/*   printf("SockCom: handleNewConnection(%d)\n", socket); */

  /* We have a new connection coming in!  We'll
     try to find a spot for it in connectlist. */
  connection = accept(*socket, NULL, NULL);
  if (connection < 0)
    {
      perror("accept");
      exit(EXIT_FAILURE);
    }
  setNonBlocking(connection);
  result = setsockopt(connection, IPPROTO_TCP, TCP_NODELAY,
       (char *) &flag, sizeof(int));
  if (result < 0)
    {
      printf("setsockopt(tcp_nodelay) failed\n");
      fflush(stdout);
      exit(1);
    }
  if(*socket == clustersock) standbysock = connection;
  for (listnum = 0; (listnum < 6) && (connection != -1); listnum ++)
    if (connectlist[listnum] == 0)
      {
        printf("\nConnection accepted:
             FD=%d; Slot=%d\n", connection,listnum);
        fflush(stdout);
        connectlist[listnum] = connection;
        connection = -1;
      }
  if (connection != -1)
    {
      /* No room left in the queue! */
      printf("\nNo room left for new client.\n");
      fflush(stdout);
      sockPuts(connection,
           "Sorry, this server is too busy. Try again later!\r\n");
      close(connection);
    }
}
void readSocks() {
  /* Current item in connectlist for for loops */
  int listnum;
  int result;
  void * outBuffer;
```

```
  int * pCode;
  int len;
  SaAmfReadinessStateT* pReadinessState;
  /* OK, now socks will be set with whatever socket(s)
     are ready for reading.  Lets first check our
     "listening" socket, and then check the sockets
     in connectlist. */
  /* If a client is trying to connect() to our listening
     socket, select() will consider that as the socket
     being 'readable'. Thus, if the listening socket is
     part of the fd_set, we need to accept a new connection. */
  if (FD_ISSET(sock, &socks))
    handleNewConnection(&sock);
  if (primary && FD_ISSET(clustersock, &socks))
    {
/*       printf("Sockcom: Ny forbindelse til clustersock!!!\n"); */
      handleNewConnection(&clustersock);

      /* Set readinessState of primary node to IN_SERVICE */
      setReadinessState(SA_AMF_IN_SERVICE);

    }
  if (!primary && FD_ISSET(clustersock, &socks))
    {
      result =  sockHandleFuncCall(clustersock);
      if (result < 0)
        {

/*        printf("Connection is closed (readSocks())\n"); */
          /*exit(1);*/
          setReadinessState(SA_AMF_IN_SERVICE);
        }
    }
  /* Now check connectlist for available data */
  /* Run through our sockets and check to see if anything
     happened with them, if so 'service' them. */
  for (listnum = 0; listnum < 6; listnum++)
    {
      if (FD_ISSET(connectlist[listnum],&socks))
        {
          result =  sockHandleFuncCall(connectlist[listnum]);
          if (result < 0)
            {
/*            printf("Connection is closed (readSocks())\n"); */

              if(primary)
                {
                  outBuffer = malloc(sizeof(int)+sizeof(SaAmfReadinessStateT));
                  memset(outBuffer,0,sizeof(int)+sizeof(SaAmfReadinessStateT));
```

```
                    pCode = outBuffer;
                    pReadinessState = outBuffer + sizeof(int);

                    *pCode = SETREMOTEREADINESSSTATE;
                    *pReadinessState = SA_AMF_IN_SERVICE;

                    len = sizeof(int)+sizeof(SaAmfReadinessStateT);

                    if ((result = sockWrite (standbysock, outBuffer, len)) == -1)
                      {
                        printf("Remotereadinessstateset unsuccesful\n");
                        exit (1);
                      }

                    sleep(30);
                    exit(0);
                  }
              }
          }
      } /* for (all entries in queue) */
}
/* This function writes a character string out to a socket.  It will
   return -1 if the connection is closed while it is trying to write. */
int sockPuts(int sockfd, char *str)
{
  return sockWrite(sockfd, str, strlen(str));
}
/* This is just like the read() system call, accept that it will make
   sure that all your data goes through the socket. */
int sockRead(int sockfd, char *buf, size_t count)
{
  size_t bytes_read = 0;
  int this_read;
  while (bytes_read < count) {
    do
      this_read = read(sockfd, buf, count - bytes_read);
    while ( (this_read < 0) && (errno == EINTR) );
    if (this_read < 0)
      return this_read;
    else if (this_read == 0)
      return bytes_read;
    bytes_read += this_read;
    buf += this_read;
  }
  return count;
}
/* This is just like the write() system call, accept that it will
   make sure that all data is transmitted. */
int sockWrite(int sockfd, char *buf, size_t count)
{
```

```
  size_t bytesSent = 0;
  int thisWrite;
  while (bytesSent < count)
    {
      do
        thisWrite = write(sockfd, buf, count - bytesSent);
      while ( (thisWrite < 0) && (errno == EINTR) );
      if (thisWrite <= 0)
        return thisWrite;
      bytesSent += thisWrite;
      buf += thisWrite;
    }
  return count;
}
/* The function opens a socket connection between the primary and the
   backup node so that checkpoint data can be sent between them. */
int openClusterSocket()
{
  /* The port number after conversion from ascport */
  int port = 9732;
  /* Bind info structure */
  struct sockaddr_in serverAddress;
  /* Used so we can re-bind to our port while a previous */
  /* connection is still in TIME_WAIT state. */
  int reuseAddr = 1;
  /* Timeout for select */
  struct timeval timeout;
  /* Number of sockets ready for reading */
  int readsocks;
  int val = 0;
  printf("Open a socket for incoming connection from standbynode....\n");
  /* Obtain a file descriptor for our "listening" socket */
  clustersock = socket(AF_INET, SOCK_STREAM, 0);
  if (clustersock < 0)
    {
      perror("cluster socket");
      exit(EXIT_FAILURE);
    }
  /* So that we can re-bind to it without TIME_WAIT problems */
  setsockopt(clustersock, SOL_SOCKET, SO_REUSEADDR,
       &reuseAddr, sizeof(reuseAddr));
  /* Set socket to non-blocking with our setnonblocking routine */
  setNonBlocking(clustersock);
  /* Get the address information, and bind it to the socket */
  memset((char *) &serverAddress, 0, sizeof(serverAddress));
  serverAddress.sin_family = AF_INET;
  serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddress.sin_port = htons(port);

  if (bind(clustersock, (struct sockaddr *) &serverAddress,
```

```
            sizeof(serverAddress)) < 0 )
    {
      perror("bind");
      close(sock);
      exit(EXIT_FAILURE);
    }

  val = listen(clustersock, 1);
  if(val!=0) {
    perror("Cannot listen");
    exit(1);
  }
  if(sock > clustersock) highsock = sock;
  else highsock = clustersock;
  FD_SET(clustersock, &socks);
  return 1;
}
int connectClusterSocket()
{
  int result, len;
  struct sockaddr_in address;
  struct hostent *host;
  char name[] = PRIMARYNODE;
  int bytes_read;
  int port = 9732;
  int flag = 1;
/*   printf("Sockcom: connect to primary node....\n"); */
  /* Create a socket connection to the primary node */
  if ((clustersock = socket (AF_INET, SOCK_STREAM, 0)) == -1)
    {
      printf("Error in cluster socket creation\n");
      exit (1);
    }
  host = gethostbyname(name);
  address.sin_family = AF_INET;
  address.sin_addr = *((struct in_addr *) host->h_addr);
  address.sin_port = htons(port);
  bzero (&(address.sin_zero), 8);
    result = connect (
       clustersock,
       (struct sockaddr *) &address,
       sizeof (address));
  if (result == -1)
    {
      printf("I'm not able to connect to primary node\n");
      printf("Errno is %d\n", errno);
      printf("Sleep 10 seconds...");
      sleep(10);
      return 0;
    }
```

```
  result = setsockopt(
       clustersock,
       IPPROTO_TCP,
       TCP_NODELAY,
       (char *) &flag, sizeof(int));
  if (result < 0)
    {
      printf("setsockopt(tcp_nodelay) failed\n");
      fflush(stdout);
      exit(1);
    }
  FD_SET(clustersock, &socks);
  return 1;
}
/* The function interprets the data that has arrived at the socket */
int sockHandleFuncCall(int sockfd)
{
  int *pCode = (int *) malloc(sizeof(int));
  void *pData;
  int bytesRead;
  int result;
  void *pOutBuffer;
 /* Case INITIALIZE */
  struct amfInitData initData;
  int handle = -1;
  struct amfAppInstance *pAmfInstance;
 /* Case FINALIZE */
 /* Case COMPREG */
  int len;
  struct amfReadinessStateSetData *pAmfReadinessStateSetData;
  struct amfReadinessStateSetData amfReadinessStateSetData;
  struct amfCompRegData regData;
  const char * resources;
  int trans = 1;
  /* Case AMFRESPONSE */
  struct amfResponseData *pAmfResponseData;
  /* Case CKPTINITIALIZE */
  struct ckptInitData ckptInitData;
  int ckptHandle = -1;
  struct ckptAppInstance *pCkptInstance;
  /* Case CKPTOPEN */
  struct ckptCheckpointOpenData openData;
  int ckptCheckpointHandle = -1;
  struct ckptCheckpointReplica* pCkptCheckpointReplica;

  /* Case CKPTWRITE */
  int * pLen;
  SaErrorT res;
  void * writeDataBuffer;
  void * bufferStart;
```

```
 SaCkptCheckpointHandleT * checkpointHandle;
 SaUint32T * numberOfElements;
 SaCkptIOVectorElementT * ioVector;
 SaCkptIOVectorElementT ioVectorElement;
 SaCkptSectionIdT sectionId;
 SaUint32T * pSaUint32T;
 SaSizeT * pSaSizeT;
 int i;
/* Case CKPTREAD */
 void * readDataBuffer;
 void * readDataBufferStart;
 SaUint32T erroneousVectorIndex;
/* Case SETREMOTEREADINESSSTATE */
 SaAmfReadinessStateT readinessState;
bytesRead = read(sockfd, pCode, sizeof(int));
 fflush(stdout);
 if(bytesRead == 0) {
   printf("Nr. 1: BytesRead == 0\n");
   fflush(stdout);
 }
 if(bytesRead == sizeof(int))
   {
     switch(*pCode)
       {
       case INITIALIZE:
         pData = (void *) malloc(sizeof(struct amfInitData));
         pAmfInstance = (struct amfAppInstance *)
                                malloc(sizeof(struct amfAppInstance));
         bytesRead = read(sockfd, pData, sizeof(struct amfInitData));
         if(bytesRead == 0) {
           printf("Nr. 2: BytesRead == 0\n");
           fflush(stdout);
         }
         initData = *((struct amfInitData*)pData);
         if(createAmfInstance(pAmfInstance &(initData.amfCallbacks),
              sockfd) != SA_OK)
           {
             exit(1);
           }
         handle = pAmfInstance->amfHandle;
         fflush(stdout);
         if ((result = sockWrite(sockfd, (char*) &handle,
              sizeof(int))) == -1)
           {
             printf("Handle sending not succesful\n");
             exit (1);
           }
/*       free(pAmfInstance); */
/*       free(pData); */
         break;
```

```
            case COMPREG:
              pData = (void *) malloc(sizeof(struct amfCompRegData));
              bytesRead = read(sockfd, pData, sizeof(struct amfCompRegData));
              if(bytesRead == 0) {
                printf("Nr. 3: BytesRead == 0\n");
                fflush(stdout);
              }
              regData = *((struct amfCompRegData*)pData);
              if(createAmfCompInstance(regData.amfHandle,
                regData.compName, regData.proxyCompName) != SA_OK)
                {
                  printf("Component registration unsuccessful\n");
                  exit(1);
                }
/*            printf("Component registration successful\n"); */
              fflush(stdout);
              if(strcmp(hb->llc_ops->get_mynodeid(hb), PRIMARYNODE) == 0 &&
                 strcmp(hb->llc_ops->node_status(hb, PRIMARYNODE), "active") == 0)
                {
                  primary = 1;
                }
              if(strcmp(hb->llc_ops->get_mynodeid(hb),STANDBYNODE) == 0 &&
                 strcmp(hb->llc_ops->node_status(hb, PRIMARYNODE), "active") != 0)
                {
                  primary = 1;
                }
              if(primary)
                {
                  /* Open a socket to which the other node can connect. */
                  /* This connection is used for checkpointing */
                  openClusterSocket();
                }
              if(!primary)
                {
                  /* Connect to the primary node to be */
                  /*able to recieve checkpoint data */
                  connectClusterSocket();
                }
/*            free(pData); */
              break;
            case RESPONSE:
/*            printf("Response mottatt\n"); */
              fflush(stdout);
              pAmfResponseData = (struct amfResponseData *)
                   malloc(sizeof(struct amfResponseData));
              bytesRead = read(sockfd, pAmfResponseData,
                   sizeof(struct amfResponseData));
              if(bytesRead == 0) {
                printf("Nr. 3,5: BytesRead == 0\n");
                fflush(stdout);
```

```
        }
        free(pAmfResponseData);
        break;
      case CKPTINITIALIZE:
        pCkptInstance = (struct ckptAppInstance *)
             malloc(sizeof(struct ckptAppInstance));
        pData = (void *) malloc(sizeof(struct ckptInitData));
        bytesRead = read(sockfd, pData, sizeof(struct ckptInitData));
        if(bytesRead == 0) {
          printf("Nr. 4: BytesRead == 0\n");
          fflush(stdout);
        }
        ckptInitData = *((struct ckptInitData*)pData);
        if(createCkptInstance(pCkptInstance, &(ckptInitData.ckptCallbacks),
           sockfd) != SA_OK)
          {
            exit(1);
          }
        ckptHandle = pCkptInstance->ckptHandle;
        fflush(stdout);
        if ((result = sockWrite(sockfd,
             (char*) &ckptHandle, sizeof(int))) == -1)
          {
            printf("CkptHandle sending not succesful\n");
            exit (1);
          }
/*      free(pData); */
/*      free(pCkptInstance); */
        break;
      case CKPTOPEN:
        pData = (void *) malloc(sizeof(struct ckptCheckpointOpenData));
        bytesRead = read(sockfd, pData,
             sizeof(struct ckptCheckpointOpenData));
        if(bytesRead == 0) {
          printf("Nr. 5: BytesRead == 0\n");
          fflush(stdout);
        }
        openData = *((struct ckptCheckpointOpenData*)pData);
        pCkptCheckpointReplica = (struct ckptCheckpointReplica*)
             malloc(sizeof(struct ckptCheckpointReplica));
        if(createCkptCheckpointReplica(pCkptCheckpointReplica,
           openData.checkpointCreationAttributes.creationFlags,
           openData.checkpointCreationAttributes.checkpointSize,
           openData.checkpointCreationAttributes.retentionDuration,
           openData.checkpointCreationAttributes.maxSections,
           openData.checkpointCreationAttributes.maxSectionSize,
           openData.checkpointCreationAttributes.maxSectionIdSize,
           openData.checkpointOpenFlags) != SA_OK)
              {
                exit(1);
```

```
                }
              ckptCheckpointHandle =
                   pCkptCheckpointReplica->ckptCheckpointHandle;
              fflush(stdout);
              if ((result = sockWrite(sockfd, (char*) &ckptCheckpointHandle,
                  sizeof(int))) == -1)
                {
                  printf("CkptCheckpointHandle sending not succesful\n");
                  exit (1);
                }
/*            free(pData); */
/*            free(pCkptCheckpointReplica); */
              break;

            case CKPTWRITE:
              pLen = (int *) malloc(sizeof(int));

              bytesRead = read(sockfd, pLen, sizeof(int));
              writeDataBuffer = (void *) malloc(*pLen-(2*sizeof(int)));
              memset(writeDataBuffer, 0, *pLen-(2*sizeof(int)));

              bufferStart = writeDataBuffer;
              bytesRead = read(sockfd, writeDataBuffer, *pLen-(2*sizeof(int)));
              if(bytesRead == 0) {
                printf("Nr. 6: BytesRead == 0\n");
                fflush(stdout);
                sleep(2);
              }
              checkpointHandle = writeDataBuffer;
              writeDataBuffer += sizeof(SaCkptCheckpointHandleT);
              numberOfElements = writeDataBuffer;
              writeDataBuffer += sizeof(SaUint32T);
              ioVector = (SaCkptIOVectorElementT *)
                  malloc(sizeof(SaCkptIOVectorElementT)*(*numberOfElements));
              for(i=0;i<(*numberOfElements);i++)
                {
                  pSaUint32T = writeDataBuffer;
                  sectionId.idLen = *pSaUint32T;
                  writeDataBuffer += sizeof(SaUint32T);
                  if(sectionId.idLen == 0)
                    {
                      sectionId.id = NULL;
                    }
                  else
                    {
                      sectionId.id = writeDataBuffer;
                    }
                  writeDataBuffer += sectionId.idLen;
                  ioVectorElement.sectionId = sectionId;
                  pSaSizeT = writeDataBuffer;
```

```
            ioVectorElement.dataSize = *pSaSizeT;
            writeDataBuffer += sizeof(SaSizeT);
            ioVectorElement.dataBuffer = writeDataBuffer;
            writeDataBuffer += ioVectorElement.dataSize;
            pSaSizeT = writeDataBuffer;
            ioVectorElement.dataOffset = *pSaSizeT;
            writeDataBuffer += sizeof(SaOffsetT);
            pSaSizeT = writeDataBuffer;
            ioVectorElement.readSize = *pSaSizeT;
            writeDataBuffer += sizeof(SaSizeT);
            ioVector[i] = ioVectorElement;
          }
        /* Skriver checkpoint lokalt */
/*      printf("Sockcom: skriver checkpoint lockalt\n"); */

        if(localCheckpointWrite(checkpointHandle, ioVector,
           numberOfElements) != SA_OK)
          {
            printf("Sockcom: localCheckpointWrite failed\n");
            fflush(stdout);
            exit(1);
          }
        if(standbysock != 0)
          {
/*          printf("Sockcom: skriver checkpoint hos standby\n"); */
            /* Skriver checkpoint hos standby */
            if(remoteCheckpointWrite(checkpointHandle, ioVector,
               numberOfElements, &standbysock) != SA_OK)
              {
                printf("Sockcom: remoteCheckpointWrite failed\n");
                fflush(stdout);
                exit(1);
              }
          }
        res = SA_OK;
        if ((result = sockWrite(sockfd, (char*) &res,
            sizeof(SaErrorT))) == -1)
          {
            printf("Checkpoint write result sending not succesful\n");
            exit (1);
          }
        free(pLen);
        free(bufferStart);
        break;

      case CKPTREMOTEWRITE:
        pLen = (int *) malloc(sizeof(int));

        bytesRead = read(sockfd, pLen, sizeof(int));
```

```
writeDataBuffer = (void *) malloc(*pLen-(2*sizeof(int)));
memset(writeDataBuffer, 0, *pLen-(2*sizeof(int)));
bufferStart = writeDataBuffer;

bytesRead = sockRead(sockfd, writeDataBuffer,
     *pLen-(2*sizeof(int)));
if(bytesRead == 0) {
  printf("Nr. 6: BytesRead == 0\n");
  fflush(stdout);
  sleep(2);
}
checkpointHandle = writeDataBuffer;
writeDataBuffer += sizeof(SaCkptCheckpointHandleT);
numberOfElements = writeDataBuffer;
writeDataBuffer += sizeof(SaUint32T);
ioVector = (SaCkptIOVectorElementT *)
   malloc(sizeof(SaCkptIOVectorElementT)*(*numberOfElements));
for(i=0;i<(*numberOfElements);i++)
  {
    pSaUint32T = writeDataBuffer;
    sectionId.idLen = *pSaUint32T;
    writeDataBuffer += sizeof(SaUint32T);
    if(sectionId.idLen == 0)
      {
        sectionId.id = NULL;
      }
    else
      {
        sectionId.id = writeDataBuffer;
      }
    writeDataBuffer += sectionId.idLen;
    ioVectorElement.sectionId = sectionId;
    pSaSizeT = writeDataBuffer;
    ioVectorElement.dataSize = *pSaSizeT;
    writeDataBuffer += sizeof(SaSizeT);
    ioVectorElement.dataBuffer = writeDataBuffer;
    writeDataBuffer += ioVectorElement.dataSize;
    pSaSizeT = writeDataBuffer;
    ioVectorElement.dataOffset = *pSaSizeT;
    writeDataBuffer += sizeof(SaOffsetT);
    pSaSizeT = writeDataBuffer;
    ioVectorElement.readSize = *pSaSizeT;
    writeDataBuffer += sizeof(SaSizeT);
    ioVector[i] = ioVectorElement;
  }
/* Skriver checkpoint lokalt */
if(localCheckpointWrite(checkpointHandle, ioVector,
   numberOfElements) != SA_OK)
  {
    printf("Sockcom: localCheckpointWrite failed\n");
```

```
        fflush(stdout);
        exit(1);
      }
    res = SA_OK;
    if ((result = sockWrite(sockfd, (char*) &res,
         sizeof(SaErrorT))) == -1)
      {
        exit (1);
      }
    free(pLen);
    free(bufferStart);
    break;
case CKPTREAD:
    pLen = (int *) malloc(sizeof(int));
    bytesRead = read(sockfd, pLen, sizeof(int));
    readDataBuffer = (void *) malloc(*pLen-(2*sizeof(int)));
    memset(readDataBuffer, 0, *pLen-(2*sizeof(int)));

    readDataBufferStart = readDataBuffer;
    bytesRead = read(sockfd, readDataBuffer, *pLen-(2*sizeof(int)));
    if(bytesRead == 0) {
      printf("Nr. 7: BytesRead == 0\n");
      fflush(stdout);
    }
    checkpointHandle = readDataBuffer;
    readDataBuffer += sizeof(SaCkptCheckpointHandleT);
    numberOfElements = readDataBuffer;
    readDataBuffer += sizeof(SaUint32T);
    ioVector = (SaCkptIOVectorElementT *)
       malloc(sizeof(SaCkptIOVectorElementT)*(*numberOfElements));
    for(i=0;i<(*numberOfElements);i++)
      {
        pSaUint32T = readDataBuffer;
        sectionId.idLen = *pSaUint32T;
        readDataBuffer += sizeof(SaUint32T);
        if(sectionId.idLen == 0)
          {
            sectionId.id = NULL;
          }
        else
          {
            sectionId.id = readDataBuffer;
          }
        readDataBuffer += sectionId.idLen;
        ioVectorElement.sectionId = sectionId;
        ioVectorElement.dataSize = 0;
        ioVectorElement.dataBuffer = NULL;
        pSaSizeT = readDataBuffer;
        ioVectorElement.dataOffset = *pSaSizeT;
        readDataBuffer += sizeof(SaOffsetT);
```

```
    ioVectorElement.readSize = 0;
    ioVector[i] = ioVectorElement;
  }
if(checkpointRead(checkpointHandle, ioVector, numberOfElements,
   &erroneousVectorIndex) != SA_OK)
  {
    printf("Sockcom: checkpointRead failed\n");
    fflush(stdout);
    exit(1);
  }
free(readDataBufferStart);
/* Sending response to client */
len = 0;
len += sizeof(int); /* len */
for(i=0;i<*numberOfElements;i++)
  {
    len += sizeof(SaUint32T); /* idLen */
    len += ioVector[i].sectionId.idLen; /* id */
    len += sizeof(SaSizeT); /* dataSize */
    len += ioVector[i].dataSize; /* dataBuffer */
    len += sizeof(SaSizeT); /* readSize */
    len += sizeof(SaUint32T); /* erroneousVectorIndex */
  }

readDataBuffer = (void *) malloc(len);
memset(readDataBuffer, 0, len);

readDataBufferStart = readDataBuffer;

memcpy(readDataBuffer, &len, sizeof(int));
readDataBuffer += sizeof(int);

for(i=0;i<*numberOfElements;i++)
  {
    memcpy(readDataBuffer, &ioVector[i].sectionId.idLen,
         sizeof(SaUint32T));
    readDataBuffer += sizeof(SaUint32T);

    if(ioVector[i].sectionId.idLen != 0)
      {
        memcpy(readDataBuffer, ioVector[i].sectionId.id,
                      ioVector[i].sectionId.idLen);
        readDataBuffer += ioVector[i].sectionId.idLen;
      }

    memcpy(readDataBuffer, &ioVector[i].dataSize,
         sizeof(SaSizeT));
    readDataBuffer += sizeof(SaSizeT);

    memcpy(readDataBuffer, ioVector[i].dataBuffer,
```

```
                    ioVector[i].dataSize);
              readDataBuffer += ioVector[i].dataSize;

              memcpy(readDataBuffer, &ioVector[i].readSize,
                    sizeof(SaSizeT));
              readDataBuffer += sizeof(SaSizeT);
              memcpy(readDataBuffer, &erroneousVectorIndex,
                            sizeof(SaUint32T));
              readDataBuffer += sizeof(SaUint32T);
            }
          if ((result = send (sockfd, readDataBufferStart, len, 0)) == -1)
            {
              printf("Server: sending readData not succesful\n");
              sleep(3);
              exit (1);
            }

          free(ioVector);
          free(pLen);
          free(readDataBufferStart);

          break;
        case SETREMOTEREADINESSSTATE:
          bytesRead = read(sockfd, &readinessState,
                           sizeof(SaAmfReadinessStateT));
          if(bytesRead == 0) {
            printf("Nr. 8: BytesRead == 0\n");
            fflush(stdout);
          }
          setReadinessState(readinessState);
          break;
        default:
          printf("No match in switch in sockHandleFuncCall\n");
          printf("pCode: %d\n", *pCode);
          printf("Sockfd: %d\nClustersock: %d\n", sockfd, clustersock);
          sleep(3);
          fflush(stdout);
          break;
        }
      free(pCode);
      return 1;
    }
  else printf("Read error in sock_getFuncCall()\n");
  free(pCode);
  return -1;
}
/* Callback which is invoked by heartbeat whenever */
/* a node in the cluster changes its status */
void llc_nstatus_callback(const char *node,
     const char * status, void* private_data)
```

```
{
  int i, j;
  SaNameT timeCompName;
  int result;
  int len;
  struct amfReadinessStateSetData *pAmfReadinessStateSetData;
  struct amfReadinessStateSetData amfReadinessStateSetData;
  void *pOutBuffer;
  int *pCode = pOutBuffer;
  if(strcmp(node, hb->llc_ops->get_mynodeid(hb)) != 0 )
    {
      if(strcmp(PRIMARYNODE, hb->llc_ops->get_mynodeid(hb)) == 0)
        {
          if(strcmp(status, "dead") == 0)
            {
              /* Lukke socketforbindelse */
              close(clustersock);
              clustersock = 0;
            }
          if(strcmp(status, "up") == 0)
            {
              /* Opprette socketforbindelse */
            }
        }
      if(strcmp(STANDBYNODE, hb->llc_ops->get_mynodeid(hb)) == 0)
        {
          if(strcmp(status, "dead") == 0)
            {
              /* Takeover */
                  for(i=0; i<5; i++)
                {
                  if(connectlist[i] != sock && connectlist[i] != 0)
                    {
                      /* Send callback til standby om at denne skal ta over. */

                      for(j=0; j<POOL_SIZE; j++)
                        {
                          if (amfPool[j] != NULL &&
                               amfPool[j]->fd == connectlist[i])
                            {
                              strcpy(timeCompName.value, "Time printer");
                              timeCompName.length = strlen(timeCompName.value) +

                              amfReadinessStateSetData.invocation = 2;
                              amfReadinessStateSetData.compName =
                                   timeCompName;
                              amfReadinessStateSetData.readinessState =
                                   SA_AMF_IN_SERVICE;

                              pOutBuffer = malloc(sizeof(int)+
```

```
                    sizeof(struct amfReadinessStateSetData));
                memset(pOutBuffer, 0, sizeof(int)+
                    sizeof(struct amfReadinessStateSetData));

                pCode = pOutBuffer;
                pAmfReadinessStateSetData =
                    pOutBuffer +sizeof(int);

                *pCode = AMFREADINESSSTATESETCALLBACK;
                *pAmfReadinessStateSetData =
                    amfReadinessStateSetData;

                len = sizeof(int)+
                    sizeof(struct amfReadinessStateSetData);

                if ((result = sockWrite(connectlist[i],
                    pOutBuffer, len)) == -1)
                  {
                    printf("Callback sending not succesful\n");
                    exit (1);
                  }

              }
            }
          }
        }
      primary = 1;
      close(clustersock);
      FD_CLR(clustersock, &socks);
      clustersock = 0;
    }
  if(strcmp(status, "up") == 0)
    {
     for(i=0; i<5; i++)
        {
          if(connectlist[i] != sock && connectlist[i] != 0)
            {
              for(j=0; j<POOL_SIZE; j++)
                {
                  if (amfPool[j] != NULL &&
                       amfPool[j]->fd == connectlist[i])
                    {
                      strcpy(timeCompName.value, "Time printer");
                      timeCompName.length =
                           strlen(timeCompName.value) + 1;

                      amfReadinessStateSetData.invocation = 2;
                      amfReadinessStateSetData.compName = timeCompName;
                      amfReadinessStateSetData.readinessState =
                           SA_AMF_OUT_OF_SERVICE;
```

```
                                    pOutBuffer = malloc(sizeof(int)+
                                         sizeof(struct amfReadinessStateSetData));
                                    memset(pOutBuffer, 0, sizeof(int)+
                                         sizeof(struct amfReadinessStateSetData));

                                    pCode = pOutBuffer;
                                    pAmfReadinessStateSetData =
                                         pOutBuffer + sizeof(int);

                                    *pCode = AMFREADINESSSTATESETCALLBACK;
                                    *pAmfReadinessStateSetData =
                                         amfReadinessStateSetData;

                                    len = sizeof(int)+
                                         sizeof(struct amfReadinessStateSetData);

                                    if ((result = sockWrite(connectlist[i],
                                         pOutBuffer, len)) == -1)
                                      {
                                        printf("Callback sending not succesful\n");
                                        exit (1);
                                      }
                                  }
                              }
                          }
                      }
                    primary = 0;
                  }
              }
        }
}
int sockCommunicate ()
{
  /* The port number after conversion from ascport */
  int port = 9733;
  /* Bind info structure */
  struct sockaddr_in serverAddress;
  /* Used so we can re-bind to our port while a previous */
  /* connection is still in TIME_WAIT state. */
  int reuseAddr = 1;
  /* Timeout for select */
  struct timeval timeout;
  /* Number of sockets ready for reading */
  int readsocks;
  int firstTime = 1;
  int val = 0;
  const char * resources;
  if(firstTime)
    {
```

```
    amfInitHandlePool();
    ckptInitHandlePool();
    firstTime = 0;
  }
/* Obtain a file descriptor for our "listening" socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0)
  {
    perror("socket");
    exit(EXIT_FAILURE);
  }
/* So that we can re-bind to it without TIME_WAIT problems */
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &reuseAddr,sizeof(reuseAddr));
/* Set socket to non-blocking with our setnonblocking routine */
setNonBlocking(sock);
/* Get the address information, and bind it to the socket */
memset((char *) &serverAddress, 0, sizeof(serverAddress));
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(port);

if (bind(sock, (struct sockaddr *) &serverAddress,
     sizeof(serverAddress)) < 0 )
  {
    perror("bind");
    close(sock);
    exit(EXIT_FAILURE);
  }
/* Set up queue for incoming connections. */
fflush(stdout);
val = listen(sock,6);
if(val!=0) {
  perror("Cannot listen");
  exit(1);
}
/* Since we start with only one socket, the listening socket, */
/* it is the highest socket so far. */
highsock = sock;
memset((char *) &connectlist, 0, sizeof(connectlist));
if(hb->llc_ops->set_nstatus_callback(
     hb, llc_nstatus_callback, NULL) == HA_FAIL)
  {
    fprintf(stderr, "Callback registration failed\n");
    fflush(stderr);
    exit(1);
  }
while (1)
  { /* Main server loop - forever */
    buildSelectList();
    timeout.tv_sec = 0;
```

```
      timeout.tv_usec = 0;
      hb->llc_ops->rcvmsg(hb, 0);
/*       sleep(1); */
      /* The first argument to select is the highest file
         descriptor value plus 1. In most cases, you can
         just pass FD_SETSIZE and you'll be fine. */
      /* The second argument to select() is the address of
         the fd_set that contains sockets we're waiting
         to be readable (including the listening socket). */
      /* The third parameter is an fd_set that you want to
         know if you can write on -- this example doesn't
         use it, so it passes 0, or NULL. The fourth parameter
         is sockets you're waiting for out-of-band data for,
         which usually, you're not. */
      /* The last parameter to select() is a time-out of how
         long select() should block. If you want to wait forever
         until something happens on a socket, you'll probably
         want to pass NULL. */
      readsocks = select(highsock+1, &socks, (fd_set *) 0,
                         (fd_set *) 0, &timeout);
      /* select() returns the number of sockets that had
         things going on with them -- i.e. they're readable. */
      /* Once select() returns, the original fd_set has been
         modified so it now reflects the state of why select()
         woke up. i.e. If file descriptor 4 was originally in
         the fd_set, and then it became readable, the fd_set
         contains file descriptor 4 in it. */
      if (readsocks < 0)
        {
          perror("select");
          exit(EXIT_FAILURE);
        }
      if (readsocks == 0)
        {
          /* Nothing ready to read, just show that we're alive */
/*        printf("."); */
/*        sleep(1); */
/*        fflush(stdout); */
/*        sleep(5); */
        }
      else
        {
          readSocks();
        }
      /* Sjekk om heartbeat kjører, hvis ikke stopp main og counterapp */
    } /* while(1) */
} /* sockCommunicate */
```

testapp_ckpt.c

```
/*****************************************************
 testapp_ckpt.c

 The file forms a user application which for testing
 purposes makes use of the AMF and CKPT. Buffers
 of 1, 10, 100, 1 000 and 10 000 bytes are
 checkpointed in 1 000 iterations. One file for each
 buffer size is created. The elapsed time for each
 checkpoint is written in the corresponding file.

 Authors: Anja Lønningen & Ingunn Lund, spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
#include "ais.h"
#include "constants.h"
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/resource.h>
extern int sockfd;
SaAmfReadinessStateT compReadinessState =
     SA_AMF_OUT_OF_SERVICE;
int number = 0;
SaCkptIOVectorElementT* createIOVector(
     void* data,
     int dataSize,
     int numberOfElements)
{
  int i;
  SaCkptIOVectorElementT element;
  SaCkptIOVectorElementT* ioVector = (SaCkptIOVectorElementT*)
       malloc(numberOfElements*sizeof(SaCkptIOVectorElementT));
  SaCkptSectionIdT sectionId = SA_CKPT_DEFAULT_SECTION_ID;
  for(i=0; i<numberOfElements; i++)
    {
      element.sectionId = sectionId;
      element.dataBuffer = data;
      element.dataSize = dataSize;
      element.dataOffset = i;
      element.readSize = 0;
      ioVector[i] = element;
    }
  return ioVector;
}
int writeFile(long * timediff, int buffersize)
{
```

```
  FILE *out;
  char filename[20];
  char buffer[5];
  sprintf(buffer, "%d", buffersize);
  strcpy(filename, RESULTS_FILE);
  strcat (filename, buffer);
  strcat (filename, ".csv");
  if ((out = fopen(filename, "a")) == NULL)
    {
      printf("Unable to open the output file");
      return 0;
    }

  fprintf(out, "%ld\n", *timediff);

  fclose(out);
  return 1;
}
int writeFileUsage(long * utime, long * stime, int buffersize)
{
  FILE *out;
  char filename[20];
  char buffer[5];
  sprintf(buffer, "%d", buffersize);
  strcpy(filename, USAGE_FILE);
  strcat (filename, buffer);
  strcat (filename, ".csv");
  if ((out = fopen(filename, "a")) == NULL)
    {
      printf("Unable to open the output file");
      return 0;
    }

  fprintf(out, "%ld, %ld\n", *utime, *stime);

  fclose(out);
  return 1;
}
void ckptTest(SaAmfHandleT * amfHandle,
     SaCkptCheckpointHandleT *checkpointHandle)
{
  int numberOfElements = 1;
  SaUint32T *erroneousVectorIndex;
  SaCkptIOVectorElementT* ioVector;
  int buffersize;
  void * buffer;
  int i,j,sum;
  struct timeval start, stop, ckptStart, ckptStop;
  unsigned long timediff;
  int who = RUSAGE_SELF;
```

```
struct rusage usageStart;
struct rusage usageStop;
long usageUTime;
long usageSTime;
int ret;
for(buffersize=1; buffersize<=10000; buffersize*=10)
  {
    buffer = (void*) malloc(buffersize);
    gettimeofday(&start, NULL);
    for(i=0; i<1000; i++)
      {
        if(i%100 == 0)
          {
            printf("Runde: %d\n", i);
            fflush(stdout);
          }
        gettimeofday(&ckptStart, NULL);
        getrusage(who, &usageStart);

        if(saCkptCheckpointWrite(checkpointHandle,
           createIOVector(buffer, buffersize, numberOfElements),
           numberOfElements, erroneousVectorIndex) != SA_OK)
          {
            printf("testapp: Error while trying to
                 call SaCkptCheckpointWrite\n");
            sleep(3);
            exit(1);
          }
        gettimeofday(&ckptStop, NULL);
        getrusage(who, &usageStop);
        timediff = (ckptStop.tv_sec*1000000+ckptStop.tv_usec) -
             (ckptStart.tv_sec*1000000+ckptStart.tv_usec);
        usageUTime = (usageStop.ru_utime.tv_sec*1000000 +
             usageStop.ru_utime.tv_usec) -
             (usageStart.ru_utime.tv_sec*1000000 +
             usageStart.ru_utime.tv_usec);
        usageSTime = (usageStop.ru_stime.tv_sec*1000000 +
             usageStop.ru_stime.tv_usec) -
             (usageStart.ru_stime.tv_sec*1000000 +
             usageStart.ru_stime.tv_usec);

        writeFile(&timediff, buffersize);
        writeFileUsage(&usageUTime, &usageSTime, buffersize);
        for(j=0; j<1000; j++)
          {
            sum += j;
          }
      }
    gettimeofday(&stop, NULL);
    /* Finn diff, skriv til fil */
```

```
      free(buffer);
     }
}
void amfReadinessStateSetCallback(SaInvocationT invocation,
     const SaNameT *compName,
     SaAmfReadinessStateT readinessState)
{
/*   printf("Counterapp: amfReadinessStateSetCallback\n"); */
  compReadinessState = readinessState;
  saAmfResponse(invocation, SA_OK);
}
int main (int argc, char* argv[])
{
  SaAmfHandleT amfHandle;
  SaAmfCallbacksT amfCallbacks;
  const SaVersionT version;
  SaCkptHandleT ckptHandle;
  SaCkptCallbacksT ckptCallbacks;
  char* string;
  fd_set readfd;
  int rc;
  SaNameT counterCompName, proxyCompName;
  SaNameT ckeckpointName;
  SaCkptCheckpointCreationAttributesT checkpointCreationAttributes;
  SaCkptCheckpointOpenFlagsT checkpointOpenFlags =
      SA_CKPT_CHECKPOINT_WRITE;
  SaTimeT timeout = 100000;
  SaCkptCheckpointHandleT checkpointHandle;
  SaUint32T *erroneousVectorIndex;
  int numberOfElements = 1;
  SaCkptIOVectorElementT* ioVector;
  int * pointer;
  /*   printf("Testapp running\n"); */
  strcpy(counterCompName.value, "Number printer");
  counterCompName.length = strlen(counterCompName.value) + 1;
  amfCallbacks.saAmfReadinessStateSetCallback =
      amfReadinessStateSetCallback;

  if(saAmfInitialize(&amfHandle, &amfCallbacks, &version) != SA_OK)
    {
      printf("Error while trying to call SaAmfInitialize\n");
      exit(1);
    }
/*   printf("I'm process %d and my handle is %d\n", getpid(), amfHandle); */

 if(SaCkptInitialize(&ckptHandle, &ckptCallbacks, &version) != SA_OK)
    {
      printf("Error while trying to call SaCkptInitialize\n");
      exit(1);
```

```
  }
 strcpy(ckeckpointName.value, "Number printer checkpoint");
 ckeckpointName.length = strlen(ckeckpointName.value) + 1;
 checkpointCreationAttributes.creationFlags = SA_CKPT_WR_ALL_REPLICAS;
 checkpointCreationAttributes.checkpointSize = 10000;
 checkpointCreationAttributes.retentionDuration = 20;
 checkpointCreationAttributes.maxSections = 1;
 checkpointCreationAttributes.maxSectionSize = 10000;
 checkpointCreationAttributes.maxSectionIdSize = sizeof(int);

if(saCkptCheckpointOpen(&ckeckpointName, &checkpointCreationAttributes,
    checkpointOpenFlags, timeout, &checkpointHandle) != SA_OK)
   {
     printf("Error while trying to call SaCkptCheckpointOpen\n");
     sleep(3);
     exit(1);
   }

if(saAmfComponentRegister(&amfHandle, &counterCompName,
    &proxyCompName) != SA_OK)
   {
     printf("Error while trying to call SaAmfComponentRegister\n");
     exit(1);
   }
 while(compReadinessState == SA_AMF_OUT_OF_SERVICE)
   {
     saAmfDispatch(&amfHandle, SA_DISPATCH_ONE);
   }

if(compReadinessState == SA_AMF_IN_SERVICE)
   {
     erroneousVectorIndex = (SaUint32T *) malloc(sizeof(SaUint32T));

     if(saCkptCheckpointRead(&checkpointHandle, ioVector =
        createIOVector(NULL, 0, numberOfElements), numberOfElements,
        erroneousVectorIndex) != SA_OK)
       {
         printf("Error while trying to call SaCkptCheckpointRead\n");
         sleep(3);
         exit(1);
       }

     if(*erroneousVectorIndex != -1)
       {
         printf("testapp: VectorIndex %d failed in ckptRead()\n",
                   *erroneousVectorIndex);
       }
   }

 ckptTest(&amfHandle, &checkpointHandle);
```

```
  return 0;
}
```

testapp_failover.c

```
/*********************************************************
 testapp_failover.c
 The file forms a user application which makes use of the
 AMF and CKPT. The program is used whenever measure-
 ments of the takeover time is wanted. Thirdprocess must
 run simultaneously. An UDP message is sent to the third
 process after a checkpoint write completes. When the
 message is recieved by the third process, the data extracted
 from the message is written in a  file together with the
 systems wall clock time. When this user application, the
 main process or the node crashes the standby node will take
 over. The takeover time may be calculated from the file
 written by the third process.
 Authors: Anja Lønningen & Ingunn Lund, spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *********************************************************/
#include "ais.h"
#include "constants.h"
#include "ckptstruct.h"
#include <stdio.h>
#include <string.h>
#include <sys/select.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <netdb.h>
extern int sockfd;
SaAmfReadinessStateT compReadinessState = SA_AMF_OUT_OF_SERVICE;
int number = 0;
int node;
int seq = 1;
SaCkptIOVectorElementT* createIOVector(void* data,
     int dataSize, int numberOfElements)
{
  int i;
  SaCkptIOVectorElementT element;
  SaCkptIOVectorElementT* ioVector = (SaCkptIOVectorElementT*)
       malloc(numberOfElements*sizeof(SaCkptIOVectorElementT));
  SaCkptSectionIdT sectionId = SA_CKPT_DEFAULT_SECTION_ID;
  for(i=0; i<numberOfElements; i++)
    {
      element.sectionId = sectionId;
      element.dataBuffer = data;
```

```
      element.dataSize = dataSize;
      element.dataOffset = i;
      element.readSize = 0;
      ioVector[i] = element;
    }
  return ioVector;
}
int writeFile(long * timediff, int buffersize)
{
  FILE *out;
  char filename[20];
  char buffer[5];
  sprintf(buffer, "%d", buffersize);
  strcpy(filename, RESULTS_FILE);
  strcat (filename, buffer);
  strcat (filename, ".csv");
  if ((out = fopen(filename, "a")) == NULL)
    {
      printf("Unable to open the output file");
      return 0;
    }

  fprintf(out, "%ld\n", *timediff);

  fclose(out);
  return 1;
}
int sendData(struct udpdata data)
{
  int udpsock;
  struct sockaddr_in address;
  struct hostent *host;
  char name[] = STANDBYNODE;
  char msg[sizeof(struct udpdata)+1];
  char seqn[sizeof(int)];
  int bytes_sent;
  int port = 9730;
  /* Create a socket connection to the third process */
  if ((udpsock = socket (AF_INET, SOCK_DGRAM, 0)) == -1)
    {
      printf("Error in cluster socket creation\n");
      exit (1);
    }
  host = gethostbyname(name);
  address.sin_family = AF_INET;
  address.sin_addr = *((struct in_addr *) host->h_addr);
  address.sin_port = htons(port);
  bzero (&(address.sin_zero), 8);
  sprintf(msg, "%d", data.nodeid);
  strcat(msg, ",");
```

```
  sprintf(seqn, "%d", data.sequencenumber);
  strcat(msg, seqn);
  bytes_sent = sendto (udpsock, msg, strlen(msg)+1, 0,
                       (struct sockaddr *) &address, sizeof (address));
  if (bytes_sent == -1)
    {
      printf("I'm not able to connect to third process\n");
      printf("Errno is %d\n", errno);
      printf("Sleep 10 seconds...");
      sleep(10);
      return 0;
    }
  return 1;
}
void ckptTest(SaAmfHandleT * amfHandle,
                      SaCkptCheckpointHandleT *checkpointHandle)
{
  int numberOfElements = 1;
  SaUint32T *erroneousVectorIndex;
  SaCkptIOVectorElementT* ioVector;
  int buffersize;
  void * buffer;
  int * pSeq;
  int i,j,sum;
  struct timeval start, stop, ckptStart, ckptStop;
  long timediff;
  struct udpdata data;
  for(buffersize=1; buffersize<=1; buffersize*=10)
    {
      buffer = (void*) malloc(buffersize+4);
      pSeq = buffer;
      gettimeofday(&start, NULL);
      for(i=0; i<1000; i++)
        {
          *pSeq = seq;
          if(i%100 == 0) printf("Runde: %d\n", i);
          gettimeofday(&ckptStart, NULL);

          if(saCkptCheckpointWrite(checkpointHandle,
             createIOVector(buffer, buffersize+4, numberOfElements),
             numberOfElements, erroneousVectorIndex) != SA_OK)
            {
              printf("Error while trying to call SaCkptCheckpointWrite\n");
              sleep(3);
              exit(1);
            }
          gettimeofday(&ckptStop, NULL);
          timediff = (ckptStop.tv_sec*1000000+ckptStop.tv_usec) -
               (ckptStart.tv_sec*1000000+ckptStart.tv_usec);
          /*Send data til 3. prosess*/
```

```
          data.sequencenumber = seq;
          data.nodeid = node;
          sendData(data);
          for(j=0; j<1000; j++)
            {
              sum += j;
            }
          seq++;
        }
      gettimeofday(&stop, NULL);
      /* Finn diff, skriv til fil */
      free(buffer);
    }
}
void amfReadinessStateSetCallback(SaInvocationT invocation,
     const SaNameT *compName,
     SaAmfReadinessStateT readinessState)
{
/*   printf("Counterapp: amfReadinessStateSetCallback\n"); */
  compReadinessState = readinessState;
  saAmfResponse(invocation, SA_OK);
}
int main (int argc, char* argv[])
{
  SaAmfHandleT amfHandle;
  SaAmfCallbacksT amfCallbacks;
  const SaVersionT version;
  SaCkptHandleT ckptHandle;
  SaCkptCallbacksT ckptCallbacks;
  char* string;
  fd_set readfd;
  int rc;
  SaNameT counterCompName, proxyCompName;
  SaNameT ckeckpointName;
  SaCkptCheckpointCreationAttributesT checkpointCreationAttributes;
  SaCkptCheckpointOpenFlagsT checkpointOpenFlags =
       SA_CKPT_CHECKPOINT_WRITE;
  SaTimeT timeout = 100000;
  SaCkptCheckpointHandleT checkpointHandle;
  SaUint32T *erroneousVectorIndex;
  int numberOfElements = 1;
  SaCkptIOVectorElementT* ioVector;
  int * pointer;
  int namelen = 10;
  char name[namelen];
  int * pSeq;
  struct udpdata data;
/*   printf("Counterapp running\n"); */
  strcpy(counterCompName.value, "Number printer");
  counterCompName.length = strlen(counterCompName.value) + 1;
```

```
amfCallbacks.saAmfReadinessStateSetCallback =
     amfReadinessStateSetCallback;

if(saAmfInitialize(&amfHandle, &amfCallbacks, &version) != SA_OK)
  {
    printf("Error while trying to call SaAmfInitialize\n");
    exit(1);
  }
if(SaCkptInitialize(&ckptHandle, &ckptCallbacks, &version) != SA_OK)
  {
    printf("Error while trying to call SaCkptInitialize\n");
    exit(1);
  }

strcpy(ckeckpointName.value, "Number printer checkpoint");
ckeckpointName.length = strlen(ckeckpointName.value) + 1;
checkpointCreationAttributes.creationFlags =
     SA_CKPT_WR_ALL_REPLICAS;
checkpointCreationAttributes.checkpointSize = 10000;
checkpointCreationAttributes.retentionDuration = 20;
checkpointCreationAttributes.maxSections = 1;
checkpointCreationAttributes.maxSectionSize = 10000;
checkpointCreationAttributes.maxSectionIdSize = sizeof(int);
if(saCkptCheckpointOpen(
   &ckeckpointName,
   &checkpointCreationAttributes,
   checkpointOpenFlags,
   timeout,
   &checkpointHandle) != SA_OK)
  {
    printf("Error while trying to call SaCkptCheckpointOpen\n");
    sleep(3);
    exit(1);
  }
if(saAmfComponentRegister(&amfHandle, &counterCompName,
   &proxyCompName) != SA_OK)
  {
    printf("Error while trying to call SaAmfComponentRegister\n");
    exit(1);
  }
gethostname(name, namelen);
if(strcmp(name, PRIMARYNODE) == 0) node = 1;
else node = 2;
while(compReadinessState == SA_AMF_OUT_OF_SERVICE)
  {
    saAmfDispatch(&amfHandle, SA_DISPATCH_ONE);
  }
if(compReadinessState == SA_AMF_IN_SERVICE)
  {
    erroneousVectorIndex = (SaUint32T *) malloc(sizeof(SaUint32T));
```

```
      if(saCkptCheckpointRead(&checkpointHandle,
         ioVector = createIOVector(NULL, 0, numberOfElements),
         numberOfElements, erroneousVectorIndex) != SA_OK)
        {
          printf("Error while trying to call SaCkptCheckpointRead\n");
          sleep(3);
          exit(1);
        }
      else
        {
          pSeq = ioVector[0].dataBuffer;
          seq = *pSeq;
          /*Send data til 3. prosess*/
          data.sequencenumber = seq;
          data.nodeid = node;
          sendData(data);
          seq++;
        }

      if(*erroneousVectorIndex != -1)
        {
          printf("testapp: VectorIndex %d failed in ckptRead()\n",
               *erroneousVectorIndex);
        }
    }

  ckptTest(&amfHandle, &checkpointHandle);
  return 0;
}
```

types.h

```
/*****************************************************
 types.h
 The file includes constants which are used to identify
 function calls transmitted by the socket connection.
 Authors: Anja Lønningen & Ingunn Lund,
 fall 2004 / spring 2005
 Contact: lonninge@stud.ntnu.no / ingunnl@stud.ntnu.no
 *****************************************************/
typedef enum {
    INITIALIZE = 1,
    FINALIZE = 2,
    COMPREG = 3,
    RESPONSE = 4,
    CKPTINITIALIZE = 5,
    CKPTFINALIZE = 6,
    CKPTOPEN = 7,
```

```
    CKPTWRITE = 8,
    CKPTREAD = 9,
    AMFREADINESSSTATESETCALLBACK = 10,
    CKPTCHECKPOINTOPENCALLBACK = 11,
    CKPTREMOTEWRITE = 20,
    CKPTREMOTERESPONSE = 21,
    SETREMOTEREADINESSSTATE = 22
} IpcCallT;
```

Tillegg B

Heartbeat konfigureringsfiler

ha.cf

```
#
#       There are lots of options in this file.  All you have to have is a set
#       of nodes listed {"node ...} one of {serial, bcast, mcast, or ucast},
#       and a value for "auto_failback".
#
#       ATTENTION: As the configuration file is read line by line,
#                  THE ORDER OF DIRECTIVE MATTERS!
#
#       In particular, make sure that the udpport, serial baud rate
#       etc. are set before the heartbeat media are defined!
#       debug and log file directives go into effect when they
#       are encountered.
#
#       All will be fine if you keep them ordered as in this example.
#
#
#       Note on logging:
#       If any of debugfile, logfile and logfacility are defined then they
#       will be used. If debugfile and/or logfile are not defined and
#       logfacility is defined then the respective logging and debug
#       messages will be loged to syslog. If logfacility is not defined
#       then debugfile and logfile will be used to log messges. If
#       logfacility is not defined and debugfile and/or logfile are not
#       defined then defaults will be used for debugfile and logfile as
#       required and messages will be sent there.
#
#       File to write debug messages to
debugfile /var/log/ha-debug
#
#
#       File to write other messages to
```

```
#
logfile /var/log/ha-log
#
#
#       Facility to use for syslog()/logger
#
logfacility     local0
#
#
#       A note on specifying "how long" times below...
#
#       The default time unit is seconds
#               10 means ten seconds
#
#       You can also specify them in milliseconds
#               1500ms means 1.5 seconds
#
#
#       keepalive: how long between heartbeats?
#
keepalive 30ms
#
#       deadtime: how long-to-declare-host-dead?
#
#               If you set this too low you will get the problematic
#               split-brain (or cluster partition) problem.
#               See the FAQ for how to use warntime to tune deadtime.
#
deadtime 120ms
#
#       warntime: how long before issuing "late heartbeat" warning?
#       See the FAQ for how to use warntime to tune deadtime.
#
warntime 20ms
#
#
#       Very first dead time (initdead)
#
#       On some machines/OSes, etc. the network takes a while to come up
#       and start working right after you've been rebooted.  As a result
#       we have a separate dead time for when things first come up.
#       It should be at least twice the normal dead time.
#
initdead 10
#
#
#       What UDP port to use for bcast/ucast communication?
#
udpport 694
#
```

```
#       Baud rate for serial ports...
#
#baud   19200
#
#       serial  serialportname ...
#serial /dev/ttyS0      # Linux
#serial /dev/cuaa0      # FreeBSD
#serial /dev/cua/a      # Solaris
#
#
#       What interfaces to broadcast heartbeats over?
#
bcast   eth0            # Linux
#bcast  eth1 eth2       # Linux
#bcast  le0             # Solaris
#bcast  le1 le2         # Solaris
#
#       Set up a multicast heartbeat medium
#       mcast [dev] [mcast group] [port] [ttl] [loop]
#
#       [dev]           device to send/rcv heartbeats on
#       [mcast group]   multicast group to join (class D multicast address
#                       224.0.0.0 - 239.255.255.255)
#       [port]          udp port to sendto/rcvfrom (set this value to the
#                       same value as "udpport" above)
#       [ttl]           the ttl value for outbound heartbeats.  this effects
#                       how far the multicast packet will propagate.  (0-255)
#                       Must be greater than zero.
#       [loop]          toggles loopback for outbound multicast heartbeats.
#                       if enabled, an outbound packet will be looped back and
#                       received by the interface it was sent on. (0 or 1)
#                       Set this value to zero.
#
#
#mcast eth0 225.0.0.1 694 1 0
#
#       Set up a unicast / udp heartbeat medium
#       ucast [dev] [peer-ip-addr]
#
#       [dev]           device to send/rcv heartbeats on
#       [peer-ip-addr]  IP address of peer to send packets to
#
#ucast eth0 192.168.1.2
#
#
#       About boolean values...
#
#       Any of the following case-insensitive values will work for true:
#               true, on, yes, y, 1
#       Any of the following case-insensitive values will work for false:
```

```
#               false, off, no, n, 0
#
#
#
#       auto_failback:  determines whether a resource will
#       automatically fail back to its "primary" node, or remain
#       on whatever node is serving it until that node fails, or
#       an administrator intervenes.
#
#       The possible values for auto_failback are:
#               on      - enable automatic failbacks
#               off     - disable automatic failbacks
#               legacy  - enable automatic failbacks in systems
#                         where all nodes do not yet support
#                         the auto_failback option.
#
#       auto_failback "on" and "off" are backwards compatible with the old
#               "nice_failback on" setting.
#
#       See the FAQ for information on how to convert
#               from "legacy" to "on" without a flash cut.
#               (i.e., using a "rolling upgrade" process)
#
#       The default value for auto_failback is "legacy", which
#       will issue a warning at startup.  So, make sure you put
#       an auto_failback directive in your ha.cf file.
#       (note: auto_failback can be any boolean or "legacy")
#
auto_failback on
#
#
#       Basic STONITH support
#       Using this directive assumes that there is one stonith
#       device in the cluster.  Parameters to this device are
#       read from a configuration file. The format of this line is:
#
#         stonith <stonith_type> <configfile>
#
#       NOTE: it is up to you to maintain this file on each node in the
#       cluster!
#
#stonith baytech /etc/ha.d/conf/stonith.baytech
#
#       STONITH support
#       You can configure multiple stonith devices using this directive.
#       The format of the line is:
#         stonith_host <hostfrom> <stonith_type> <params...>
#         <hostfrom> is the machine the stonith device is attached
#              to or * to mean it is accessible from any host.
#         <stonith_type> is the type of stonith device (a list of
```

```
#              supported drives is in /usr/lib/stonith.)
#         <params...> are driver specific parameters.  To see the
#              format for a particular device, run:
#           stonith -l -t <stonith_type>
#
#
#       Note that if you put your stonith device access information in
#       here, and you make this file publically readable, you're asking
#       for a denial of service attack ;-)
#
#       To get a list of supported stonith devices, run
#               stonith -L
#       For detailed information on which stonith devices are supported
#       and their detailed configuration options, run this command:
#               stonith -h
#
#stonith_host *     baytech 10.0.0.3 mylogin mysecretpassword
#stonith_host ken3  rps10 /dev/ttyS1 kathy 0
#stonith_host kathy rps10 /dev/ttyS1 ken3 0
#
#       Watchdog is the watchdog timer.  If our own heart doesn't beat for
#       a minute, then our machine will reboot.
#       NOTE: If you are using the software watchdog, you very likely
#       wish to load the module with the parameter "nowayout=0" or
#       compile it without CONFIG_WATCHDOG_NOWAYOUT set. Otherwise even
#       an orderly shutdown of heartbeat will trigger a reboot, which is
#       very likely NOT what you want.
#
#watchdog /dev/watchdog
#
#       Tell what machines are in the cluster
#       node    nodename ...    -- must match uname -n
node    stud2104
node    stud2103
#
#       Less common options...
#
#       Treats 10.10.10.254 as a psuedo-cluster-member
#       Used together with ipfail below...
#
#ping 10.10.10.254
#
#       Treats 10.10.10.254 and 10.10.10.253 as a psuedo-cluster-member
#       called group1. If either 10.10.10.254 or 10.10.10.253 are up
#       then group1 is up
#       Used together with ipfail below...
#
#ping_group group1 10.10.10.254 10.10.10.253
#
#       Processes started and stopped with heartbeat.  Restarted unless
```

```
#               they exit with rc=100
#
#respawn userid /path/name/to/run
#respawn hacluster /usr/lib/heartbeat/ipfail
#
#       Access control for client api
#               default is no access
#
#apiauth client-name gid=gidlist uid=uidlist
#apiauth ipfail gid=haclient uid=hacluster
apiauth hbtest gid=haclient uid=hacluster,ingunn,root
apiauth ais gid=haclient uid=hacluster,ingunn,root
############################
#
#       Unusual options.
#
############################
#
#       hopfudge maximum hop count minus number of nodes in config
#hopfudge 1
#
#       deadping - dead time for ping nodes
#deadping 30
#
#       hbgenmethod - Heartbeat generation number creation method
#               Normally these are stored on disk and incremented as needed.
#hbgenmethod time
#
#       realtime - enable/disable realtime execution (high priority, etc.)
#               defaults to on
#realtime off
#
#       debug - set debug level
#               defaults to zero
#debug 1
#
#       API Authentication - replaces the fifo-permissions-based system of the pa
#
#
#       You can put a uid list and/or a gid list.
#       If you put both, then a process is authorized if it qualifies under eith
#       the uid list, or under the gid list.
#
#       The groupname "default" has special meaning.  If it is specified, then
#       this will be used for authorizing groupless clients, and any client grou
#       not otherwise specified.
#
#apiauth        ipfail uid=hacluster
#apiauth ccm uid=hacluster
#apiauth ping gid=haclient uid=alanr,root
```

```
apiauth default gid=haclient
# message format in the wire, it can be classic or netstring, default is classic
#msgfmt  netstring
```

haresources

```
#
#       This is a list of resources that move from machine to machine as
#       nodes go down and come up in the cluster.  Do not include
#       "administrative" or fixed IP addresses in this file.
#
# <VERY IMPORTANT NOTE>
#       The haresources files MUST BE IDENTICAL on all nodes of the cluster.
#
#       The node names listed in front of the resource group information
#       is the name of the preferred node to run the service.  It is
#       not necessarily the name of the current machine.  If you are running
#       auto_failback ON (or legacy), then these services will be started
#       up on the preferred nodes - any time they're up.
#
#       If you are running with auto_failback OFF, then the node information
#       will be used in the case of a simultaneous start-up, or when using
#       the hb_standby {foreign,local} command.
#
#       BUT FOR ALL OF THESE CASES, the haresources files MUST BE IDENTICAL.
#       If your files are different then almost certainly something
#       won't work right.
# </VERY IMPORTANT NOTE>
#
#
#       We refer to this file when we're coming up, and when a machine is being
#       taken over after going down.
#
#       You need to make this right for your installation, then install it in
#       /etc/ha.d
#
#       Each logical line in the file constitutes a "resource group".
#       A resource group is a list of resources which move together from
#       one node to another - in the order listed.  It is assumed that there
#       is no relationship between different resource groups.  These
#       resource in a resource group are started left-to-right, and stopped
#       right-to-left.  Long lists of resources can be continued from line
#       to line by ending the lines with backslashes ("\").
#
#       These resources in this file are either IP addresses, or the name
#       of scripts to run to "start" or "stop" the given resource.
#
#       The format is like this:
#
```

```
#node-name resource1 resource2 ... resourceN
#
#
#       If the resource name contains an :: in the middle of it, the
#       part after the :: is passed to the resource script as an argument.
#       Multiple arguments are separated by the :: delimeter
#
#       In the case of IP addresses, the resource script name IPaddr is
#       implied.
#
#       For example, the IP address 135.9.8.7 could also be represented
#       as IPaddr::135.9.8.7
#
#       THIS IS IMPORTANT!!     vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
#
#       The given IP address is directed to an interface which has a route
#       to the given address.  This means you have to have a net route
#       set up outside of the High-Availability structure.  We don't set it
#       up here -- we key off of it.
#
#       The broadcast address for the IP alias that is created to support
#       an IP address defaults to the highest address on the subnet.
#
#       The netmask for the IP alias that is created defaults to the same
#       netmask as the route that it selected in in the step above.
#
#       The base interface for the IPalias that is created defaults to the
#       same netmask as the route that it selected in in the step above.
#
#       If you want to specify that this IP address is to be brought up
#       on a subnet with a netmask of 255.255.255.0, you would specify
#       this as IPaddr::135.9.8.7/24 .
#
#       If you wished to tell it that the broadcast address for this subnet
#       was 135.9.8.210, then you would specify that this way:
#               IPaddr::135.9.8.7/24/135.9.8.210
#
#       If you wished to tell it that the interface to add the address to
#       is eth0, then you would need to specify it this way:
#               IPaddr::135.9.8.7/24/eth0
#
#       And this way to specify both the broadcast address and the
#       interface:
#               IPaddr::135.9.8.7/24/eth0/135.9.8.210
#
#       The IP addresses you list in this file are called "service" addresses,
#       since they're they're the publicly advertised addresses that clients
#       use to get at highly available services.
#
#       For a hot/standby (non load-sharing) 2-node system with only
```

```
#       a single service address,
#       you will probably only put one system name and one IP address in here.
#       The name you give the address to is the name of the default "hot"
#       system.
#
#       Where the nodename is the name of the node which "normally" owns the
#       resource.  If this machine is up, it will always have the resource
#       it is shown as owning.
#
#       The string you put in for nodename must match the uname -n name
#       of your machine.  Depending on how you have it administered, it could
#       be a short name or a FQDN.
#
#-------------------------------------------------------------------
#
#       Simple case: One service address, default subnet and netmask
#               No servers that go up and down with the IP address
#
#just.linux-ha.org      135.9.216.110
stud2104                129.241.102.86 Ais
#
#-------------------------------------------------------------------
#
#       Assuming the adminstrative addresses are on the same subnet...
#       A little more complex case: One service address, default subnet
#       and netmask, and you want to start and stop http when you get
#       the IP address...
#
#just.linux-ha.org      135.9.216.110 http
#-------------------------------------------------------------------
#
#       A little more complex case: Three service addresses, default subnet
#       and netmask, and you want to start and stop http when you get
#       the IP address...
#
#just.linux-ha.org      135.9.216.110 135.9.215.111 135.9.216.112 httpd
#-------------------------------------------------------------------
#
#       One service address, with the subnet, interface and bcast addr
#       explicitly defined.
#
#just.linux-ha.org      135.9.216.3/28/eth0/135.9.216.12 httpd
#
#-------------------------------------------------------------------
#
#       An example where a shared filesystem is to be used.
#       Note that multiple aguments are passed to this script using
#       the delimiter '::' to separate each argument.
#
#node1  10.0.0.170 Filesystem::/dev/sda1::/data1::ext2
```

```
#
#       Regarding the node-names in this file:
#
#       They must match the names of the nodes listed in ha.cf, which in turn
#       must match the ‘uname -n‘ of some node in the cluster.  So they aren’t
#       virtual in any sense of the word.
#
```

authkeys

```
#
#       Authentication file.  Must be mode 600
#
#
#       Must have exactly one auth directive at the front.
#       auth    send authentication using this method-id
#
#       Then, list the method and key that go with that method-id
#
#       Available methods: crc sha1, md5.  Crc doesn’t need/want a key.
#
#       You normally only have one authentication method-id listed in this file
#
#       Put more than one to make a smooth transition when changing auth
#       methods and/or keys.
#
#
#       sha1 is believed to be the "best", md5 next best.
#
#       crc adds no security, except from packet corruption.
#               Use only on physically secure networks.
#
auth 1
1 crc
2 sha1 key_for_sha1
3 md5 key_for_md5
```