Överföring av program från värddator till målmiljö

(HS-IDA-EA-99-101)

Nicklas Bergfeldt (a96nicbe@ida.his.se)

Institutionen för datavetenskap
Högskolan i Skövde, Box 408
S-54128 Skövde, SWEDEN

Examensarbete på det datavetenskapliga programmet under vårterminen 1999.

Handledare: Joakim Eriksson

Överföring av program från värddator till målmiljö

Examensrapport inlämnad av Nicklas Bergfeldt till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för Datavetenskap.

1999-06-18

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

Signerat: _____

Överföring av program från värddator till målmiljö

Nicklas Bergfeldt (a96nicbe@ida.his.se)

Sammanfattning

Detta arbete tar upp olika överföringstekniker för att uppdatera ett program på en målmiljö. De tre överföringstekniker som analyseras är moduluppdelning, patchning och komprimering. En litteraturstudie har genomförts för att ta fram för- och nackdelar med de olika teknikerna och för att beskriva hur de fungerar. En praktisk applicering av överföringsteknikerna har även gjorts på ett specifikt PLC-system för att se vad dessa kan göra för uppdateringstiden i denna miljö. Resultatet av arbetet visade att moduluppdelning är att föredra i de fall där uppdelning av programmet är möjlig. Patchning är däremot mest lämpad då varje uppdatering av programmet är mycket liten och det således inte skiljer mycket mellan de olika versionerna av programmet. Komprimering är att föredra då det ofta är frågan om stora modifieringar i programmet eftersom komprimering ger en näst intill konstant vinst sett ur procentuell synvinkel, även om denna vinst endast är runt 50%.

Nyckelord: Datakommunikation, Prestanda, PLC (Programmable Logic Interface)

Innehållsförteckning

1 Introduktion

Dagens samhälle blir allt mer datoriserat. Denna datorisering har en given fördel med avseende på att effektiviteten ökar, men samtidigt måste dessa system underhållas och uppdateras då fel upptäcks, eller då ny funktionalitet önskas i de olika systemen. En uppdatering av detta slag går ofta till så att det nya programmet utvecklas och görs färdigt på en utvecklingsdator. Anledningen till att programmet utvecklas på ett system och sedan överförs till ett annat efter det att programmet är färdigt är att i de flesta fall är detta den enda möjligheten. Oftast finns inte någon möjlighet att utveckla programmet på den miljö som programmet skall köras på eftersom funktionaliteten hos målmiljön endast brukar räcka till att köra det färdiga programmet. Dessutom är möjligheterna att testa programmet på en värddator värdefulla ur flera avseenden. De simuleringar som kan utföras på en värddator kan ge information om hur väl programmet är gjort och om det finns några allvarliga fel i programmet. Med hjälp av dessa simuleringar och tester på en värddator så kan onödigt arbete undvikas och riskerna minskas då det inte gör så mycket om det blir fel i en simulering. Det skulle vara mycket värre om felet hände i verkligheten och ett flertal personer till exempel dödas av en skenande robot.

Vid uppdateringen ansluts datorn med det nya programmet till det system som skall uppdateras. Ett sådant scenario brukar kallas en värd–målmiljö där värddatorn utgör den del som utvecklingen sker på och därifrån som alla program och uppdateringar skickas. Exempel på värddator kan vara en vanlig persondator som finns i de flesta hem. Målmiljön är det system som sköter själva processen och kan till exempel vara en telefonväxel eller en industrirobot. Eftersom utvecklingen av programmet sker på värddatorn och användandet av programmet sker i målmiljön så ger detta att en kommunikation mellan värddator och målmiljö måste ske. Denna kommunikation skall överföra all den data som krävs för att uppdateringen skall lyckas, varför det är viktigt att överföringen sker på ett tillfredsställande sätt. Denna överföring av data kan innebära att en liten modifiering utförs eller att hela systemet kommer att bytas ut och helt ny funktionalitet kommer att uppnås.

Den överföring som äger rum kan ta olika lång tid beroende på mängden data som behöver överföras och vilket transmissionsmedium som används. I vissa situationer tar överföringen för lång tid, vilket ger att det finns tid att spara om ett effektivare utnyttjande av de tillgängliga resurserna kan göras. Om en målmiljö skall uppdateras med ett nytt program och det nya programmet är av storleken 5 Mbyte så är den tid som kommer att gå åt givetvis beroende på vilket transmissionsmedium som används. Om en 10 Mbits/sek ethernetanslutning används så kommer överföringen endast att ta 4 sekunder. Men om det däremot är så att en RS-232 förbindelse på 38,2 kbits/sek används så tar överföringen hela 18 minuter att genomföra. Detta exempel belyser att det kan variera oerhört på vilken typ av transmissionsmedium som väljs. Om det däremot hade varit frågan om ett program på 5 Kbyte hade tiden varit 4 hundradels sekunder med ethernet och 1 sekund med RS-232 för att överföra hela programmet. Här syns att vid små datamängder finns det ofta inte mycket tid att hämta, men i takt med att datamängden ökar så ökar alltså tiden det tar att överföra allting.

Många processer som utförs ute i industrin är beroende av varandra. Det är ofta en följd av processer som måste komma i en viss ordning och det kan även röra sig om

vissa tidsbegränsningar. Ett exempel på detta kan vara förpackningsmaskinen i ett livsmedelsverk där det inte får vara så att livsmedlen ligger oförpackade längre än en timme. Ytterligare ett exempel är att det är många kompressorer som måste ha ett visst tryck för att kunna starta maskinen och en viss temperatur på vattnet skall hållas. Många sådana här faktorer kan spela in och resultera i att det skulle bli förödande om denna maskin tas ur drift på grund av en uppdatering av mjukvaran, för om det sker så tar det alldeles för lång tid att starta maskinen igen. Vid dessa tillämpningar är det alltså mycket önskvärt att kunna undvika att stanna eller avbryta målmiljön under tiden som en uppdatering sker.

Att utföra en snabb uppdatering är inte bara önskvärt då maskinen som skall uppdateras måste stannas under uppdatering, utan den snabba uppdateringen kan även vara värdefull i andra fall. Till exempel kan detta vara fallet då ett system som skall uppdateras endast har en begränsad funktionalitet under tiden som uppdatering pågår och därför bör systemet så fort som möjligt bli klart för att tas i drift. Ytterligare ett exempel där en snabb uppdatering är önskvärd kan vara om felet som skall rättas till resulterar i förluster per tidsenhet. I dessa fall är en snabb uppdatering väsentlig eftersom varje sekund kan vara dyrbar.

Detta arbete går ut på att minimera den överföringstid som krävs vid uppdatering av en målmiljö. Till överföringstiden räknas här även den tid som det tar att förbereda data som skall skickas och även det eventuella efterarbete som krävs i målmiljön innan systemet kan tas i drift. Därtill kommer den tid det tar att överföra datamängden som behövs för att uppdatera målmiljön. Observera här att den datamängd som överförs inte nödvändigtvis behöver vara ett komplett system som bara är att köra igång på målmiljön, utan att det kan vara frågan om en liten del som skall användas till att modifiera målmiljön så att en uppdatering kan ske utan att allting överförs. Summan av dessa tider kommer härefter att refereras till som den totala överföringstiden. I detta arbete identifieras några olika tekniker som skulle kunna hjälpa till med att förkorta den totala överföringstiden. Dessa tekniker beskrivs och en jämförande studie presenteras därefter som visar på de olika karakteristiska dragen hos varje överföringsteknik. Överföringsteknikerna applicerades även på ett praktiskt fall och slutsatserna från den teoretiska studien fick ligga som beslutsunderlag i vissa frågor.

Eftersom rapporten tar upp olika för- och nackdelar som karakteriserar de olika överföringsteknikerna kan den tjäna som beslutshjälp vid val av överföringsteknik och uppdateringsmetod för ett system som uppdateras via en kommunikationslänk.

2 Bakgrund och begrepp

Detta kapitel förklarar olika termer och begrepp som förekommer i denna rapport.

2.1 Värddator

Värddatorn är den maskin som har det program som skall överföras till målmiljön. Det är ofta så att utvecklingen av den nya programvaran sker på värddatorn men detta är inget krav. Värddatorns uppgift är bland annat att överföra det nya programmet till målmiljön på ett säkert sätt, det är inte tvunget att utvecklingen av programmet sker på värddatorn. Utöver utvecklingen av den nya programvaran är det också ofta så att det är på värddatorn som simuleringar och tester utförs.

2.2 Målmiljö

Målmiljön är det system som skall ta emot det nya programmet från värddatorn. Med andra ord är det målmiljön som är den maskin som skall uppdateras med det nya programmet. Målmiljön kan till exempel vara ett trafikljus eller en robot. Det är på målmiljön som allt jobb utförs efter det att programmet har blivit överfört och startat. Vid till exempel ett trafikljus är det målmiljön som känner av när det kommer in bilar från olika håll och ser till så att det inte blir grönt vid fel tidpunkt.

2.3 Kommunikation

Kommunikation mellan datorer kan ske på olika sätt, överföring av data kan till exempel ske via en kabel som kopplar samman datorerna, eller också kan någon form av trådlös kommunikation användas. Oavsett om en kabel eller någon form av trådlös kommunikation används så måste specifika överföringsprotokoll användas som ser till att data överförs korrekt. En standard som har använts mycket och som fortfarande används är RS-232 (Campbell, 1990). Denna standard definierar inte bara hur starka signalerna skall vara vid överföringen utan bestämmer även över hur kontaktdonen skall se ut och vilken pin-konfiguration som skall föreligga.

Ett exempel är att använda sig av kabelbaserad kommunikation via RJ45-kabel (tvinnat trådpar) och på detta använda sig av protokollet ethernet (vanlig benämning på IEEE 802.3, även om de inte tekniskt är samma standard (Hancock, 1996)).

2.4 Moduluppdelning

Moduluppdelning innebär att programmet delas upp i flera delar, så kallade moduler (Rumbaugh, 1991). Dessa moduler har en klart preciserad funktionalitet och det kan till exempel vara frågan om enskilda funktioner eller en grupp av funktioner som samarbetar på ett nära sätt. För att få ett komplett program används flera moduler tillsammans, så att en vidare funktionalitet uppnås än om bara en av modulerna används. Modulerna arbetar tillsammans och utgör det kompletta programmet.

2.5 Patchning

Patchning är en teknik som går ut på att få fram minsta möjliga förändring vid en uppdatering. Patchningstekniken bygger på det faktum att det ofta vid uppdateringar är så att det inte är hela koden som skrivs om eller byts ut, utan att det ofta är frågan om mindre modifieringar. Dessa modifieringar sorteras ut så att endast en liten

datamängd blir kvar, det som har förändrats från det tidigare programmet. Det är denna lilla datamängd som utgör själva uppdateringen och det är endast detta som behöver överföras till målmiljön. På målmiljön finns redan allt det andra som inte har förändrats så nu behövs bara det som modifierats för att få en komplett uppdatering.

2.6 Komprimering

Att använda komprimering vid uppdatering av en målmiljö innebär att den data som skall skickas någonstans först minskas i storlek med hjälp av komprimerings-algoritmer. Dessa algoritmer försöker att minska den datamängd som skall skickas genom att till exempel leta efter likheter och symmetri, som i sin tur kan beskrivas kortare än vad som gjorts i originalprogrammet. När dessa algoritmer har gått igenom den data som skall skickas så har en datamängd skapats som är mindre än originalet men som ändå innehåller samma information. Det är denna förminskade, komprimerade, datamängd som sedan skickas till målmiljön. För att ge en tydligare förståelse för vad komprimeringsalgoritmer gör och hur de kan fungera ges följande exempel på en enkel algoritm för komprimering av textfiler. Sök i texten efter sekvenser av tecken där samma tecken upprepas minst tre gånger. Byt sedan ut denna sekvens mot tre tecken av detta slag följt av en siffra som talar om hur många tecken som det skall vara totalt här. Som exempel kan nämnas att sekvensen ”abbbbbbbba” blir ”abbb8a” komprimerat. Just i detta fallet komprimeras texten från 10 till 6 tecken, vilket nästan är en halvering av storleken. Även om denna algoritm är mycket enkel så finns det andra komprimeringsalgoritmer som är betydligt mer komplicerade. Dock har de alla en gemensam grundtanke: att förkorta och trycka ihop data till en mindre mängd som fortfarande skall kunna ge den ursprungliga datan.

3 Problembeskrivning

Problemet är att minska överföringstiden vid uppdatering av program i ett målsystem. Denna överföring kan ta ansenlig tid då mycket data (stora program) skall överföras eller då ett långsamt transmissionsmedium används. Som tidigare visats så är detta ett problem som är viktigt att lösa. Vidare är det önskvärt att undvika stopp i målmiljön under tiden som denna överföring sker.

3.1 Problemprecisering

Detta arbete fokuserar på följande tre alternativa tekniker gällande överföring av program från en värddator till ett målsystem. Dessa tre tekniker är de enda som identifierats för detta arbete.

- Moduluppdelning - Överföring av den modul som modifierats.

  I och med att objektorienterad programvaruutveckling har vuxit sig så stort som den har gjort på senare tid har moduluppdelning av programvara blivit ett vanligt koncept.

- Patchning - Överföring av endast det som modifierats.

  Anledningen till att patchning är med i denna undersökning är att det idag är en mycket vanlig teknik att uppdatera programvara med. Fördelen med denna teknik är att det ofta blir lite som behöver överföras varje gång vilket innebär att minskningen av nedladdningstiden kan bli signifikant.

- Komprimering - Överföring av programmet i komprimerad form.

3.2 Problemavgränsning

Att kombinera de olika överföringsteknikerna är möjligt. Till exempel skulle komprimering av en patch till en modul vara fullt möjlig och det är också mycket troligt att kombinationer av olika slag kan visa sig mycket framgångsrika vad gäller minskande av den totala uppdateringstiden. På grund av att detta arbete var tidsbegränsat analyserades inte fler överföringstekniker (även om inga andra identifierats) och inte heller har olika kombinationer av överföringstekniker undersökts.

I många undersökningar är det mycket givande att göra en omfattande praktisk studie och således ta upp många olika variationer av program och överföringar. Då detta arbete är mycket tidskrävande har endast en begränsad praktisk studie genomförts.

3.3 Diskussion: alternativa lösningsmetoder

Ett alternativ till att använda överföringstekniker som optimerar resursutnyttjandet kan vara att öka bandbredden och på så sätt få snabbare överföring. Detta är visserligen en lösning på problemet men lösningen innebär egentligen bara att problemet skjuts på framtiden. Visst är det möjligt att öka bandbredden men det finns hela tiden en övre gräns för hur snabbt det kan gå: ljushastigheten är än så länge den maximala hastighet vi kan utnyttja vid överföring. Däremot är det inte så att storleken är begränsad på något sätt, det går hela tiden att överföra mer information. Detta

kombinerat med det faktum att kraven på datorer ständigt och snabbt ökar ger att problemet hela tiden kommer att kvarstå, varför ett optimeringsproblem som detta är ett intressant problem att lösa.

I vissa fall kan en ökning av bandbredden vara en fullt tillräcklig metod att lösa problemet temporärt på, men beaktning bör alltid tas till det som nämndes tidigare. Kraven ökar konstant och det blir hela tiden större datamängder att överföra varför den nya hastigheten inte heller kommer att räcka till. Frågan är bara hur snart som den nya hastigheten blir för långsam igen och en ny uppdatering måste göras och om det då kanske är lönt att titta på alternativa överföringstekniker som utnyttjar bandbredden effektivare.

4 Metod

Överförandet av ett uppdaterat program från värddator till målmiljö skall utföras. Denna överföring skall ske så snabbt som möjligt och om möjligheten finns så är det mycket önskvärt att kunna undvika driftsstopp i systemet under tiden som uppdateringen utförs. De tekniker som undersöktes i denna rapport var modul-uppdelning, patchning och komprimering.

4.1 Genomförande av jämförande studie

De tre teknikerna som ställdes upp under problempreciseringen i kapitel 3.1 utvärderades utefter följande sex kriterier. Dessa är valda så att de innefattar arbete innan, under och efter överföring av det nya programmet och således representeras hela uppdateringskedjan. Med uppdateringskedja avses alla procedurer från det att det nya, färdigutvecklade programmet har tagits fram tills det att programmet körs på målmiljön. Eftersom teknikerna inte utför lika mycket arbete vid varje del av uppdateringskedjan måste hela denna betraktas, varför alla sex kriterier har varit med i den jämförande studien.

- Krav på målmiljö

  I detta kriterium kommer de krav på målmiljön som de olika teknikerna har att väga in. Vad som behandlats här är de krav på målmiljön i form av att speciell hårdvara eller mjukvara måste finnas för att kunna utnyttja de olika teknikerna. Till exempel kan det vara nödvändigt att använda sig av ett specifikt operativ-system för att kunna använda sig av någon teknik. Ett annat krav som kan föreligga är att det måste finnas stöd för vissa specifika funktioner i operativ-systemet.

- Mängden förarbete innan överföring av data kan ske

  Detta är det arbete som krävs för att veta vilken data som skall överföras och andra eventuella extra-arbeten som krävs innan överföring kan börja. Ett tydligt exempel är vid komprimering, då programmet som skall överföras först måste komprimeras och det är just komprimeringen av programmet som blir förarbete i ett sådant fall.

- Mängden data som behöver överföras

  Den datamängd som kommer att tas med i detta kriterium är endast den datamängd som kommer att överföras från värddator till målmiljö för att göra uppdateringen möjlig. Detta kan till exempel vara fallet vid komprimering, där den komprimerade datan då oftast blir mindre än originaldatan. I ett sådant fall behöver endast den komprimerade datan överföras för att uppdatera målmiljön, vilket medför att den överförda datan är mindre än den mängd data som fås på målmiljön efter dekomprimering.

- Mängden efterarbete av data innan start på målmiljö kan ske.

  I vissa fall är det nödvändigt att utföra en del arbete på målmiljön innan start kan ske. Till exempel kan det vara att datan som skickas över inte är den faktiska data som skall köras, utan den datamängd som skickas över är bara en del i det hela programmet. Om så är fallet måste en del arbete ske innan uppdateringen är

färdig. Observera att mängden efterarbete mycket väl kan påverkas av huruvida undvikandet av avbrott önskas eller ej. Detta kriterium kommer endast att se till de fall då undvikandet av avbrott ej är nödvändigt (se nedanstående kriterium).

- Möjlighet att undvika avbrott.

  Som tidigare nämnts är undvikandet av avbrott under tiden som uppdatering sker ett önskvärt kriterium. En analys har genomförts där det visar sig hur pass väl de olika teknikerna lämpar sig för att undvika ett maskinstopp under tiden som en uppdatering utförs. För att kunna uppdatera systemet under tiden som det är i gång kan det krävas en del extra-arbete på målmiljön innan uppdateringen är klar. Det kan alltså innebära arbete utöver vad som kommit fram i föregående kriterium.

  Detta kriterium belyser vad som ytterligare krävs för att kunna undvika stopp av målsystemet under tiden som uppdateringen sker.

- Storleksberoende.

  Som tidigare nämnts så är den totala överföringstiden mycket beroende på vilken mängd data som skall överföras. I detta kriterium kommer hänsyn att tas till tre olika kategorier av storleken på datamängden som skall överföras. Dessa tre kategorier visar på hur förhållandena kan variera då det är frågan om stora eller små system som skall uppdateras på målmiljön. Rapporten tar upp de fall som denna variation påverkar överföringstekniken och således vinsten med respektive teknik.

  System: Denna kategori utgör ett komplett system som kan innefatta flera olika program och med detta en komplett struktur över flera filer och kataloger. Detta är den största datamängd som överförs.

  Program: Här innefattas endast ett program eller en modul i ett program. Det som överförs är alltså endast ett program eller en modul till ett program och storleken kan således vara relativt liten, jämfört med ett system.

  Byte: Detta innebär att en mycket liten datamängd överförs till målmiljön. Det kan till exempel vara vid en variabeluppdatering som en så liten datamängd behöver överföras.

På grund av att en teoretisk jämförelse har genomförts så har en omfattande litteraturstudie använts för att få fram vad som karakteriserar de olika överföringsteknikerna. Efter varje genomgången teknik sammanfattas resultaten från litteraturstudien och det är i denna sammanfattning som alla sex kriterier sammanställs så att ett helhetsbetyg kan ges för de olika teknikerna. I sammanfattningen ges även en analys av den potentiella vinst som kan göras om respektive teknik används.

4.2 Praktisk applicering av teorier

Som stöd för de resultat som ges under den teoretiska fasen av arbetet har även en praktisk analys gjorts av ett specifikt system. Systemet i fråga heter BiFas UHS och är utvecklat av Binär Elektronik AB. Den praktiska delen av arbetet undersöker huruvida några av ovanstående tekniker kan appliceras på BiFas UHS systemet.

5 Jämförande studie av överföringstekniker

Detta kapitel tar upp de resultat som framkommit under den litteraturstudie som utförts. De tre teknikerna redovisas först grupperade efter varje teknik och därefter följer en sammanfattning av studien. Sist i kapitlet summeras och värderas alla överföringstekniker gentemot varandra.

Varje överföringsteknik inleds med en bakgrund och en allmän information om hur tekniken i fråga fungerar. Därefter analyseras tekniken gentemot de kriterier som ställts upp.

5.1 Moduluppdelning

Moduluppdelning går ut på att dela upp programmet i flera separata delar, så kallade moduler. En modul kan ses som en instans av ett objekt i objektorienterad programmering. Denna liknelse kan dras då modulen har många av de karakteristiska dragen för just en instans av ett objekt (Burns and Wellings, 1996). Moduluppdelning av ett program kan ses som en variant av objektorienterad programmering[1] där varje modul motsvaras av ett objekt och då programmet startas skapas en instans av de objekt, moduler, som behövs. I objektorienterad programmering kan flera instanser av samma objekt skapas och detta kan uppnås i moduluppdelade program också genom att använda flera trådar, processer, på en miljö som kan hantera detta. Moduler har ett klart definierat gränssnitt som preciserar hur modulerna skall användas och på grund av att modulerna byggs upp som de gör erhålls en inkapsling som innebär att all data som finns i modulen är skyddad och inget kan komma åt denna data utom via det gränssnitt som är uppsatt (Rumbaugh m.fl., 1991). Detta är viktiga egenskaper hos moduler, just att de har inkapsling och ett klart definierat gränssnitt. Dessa egenskaper är väsentliga därför att de resulterar i en lös sammankoppling (eng. loose coupling (Pressman, 1997)) mellan de olika modulerna. Vid tillräckligt lös samman-koppling behöver endast den modul som förändrats bytas ut. Om däremot denna lösa sammankoppling ej uppnåtts kan flera moduler behöva bytas trots att det egentligen bara är en modul som modifierats. Detta kan hända då det fördefinierade gränssnittet inte används till att hämta eller manipulera data inuti modulerna utan bakvägar istället används. Alltså är inkapslingen och gränssnittet mycket viktiga egenskaper och de är även nödvändiga för att uppnå önskad funktionalitet med moduluppdelning.

Moduluppdelning kan även uppnås genom att dela in målmiljön i flera program och använda varje program som en modul. I vissa fall kan det vara svårt att skilja på ett program och en modul, eftersom det faktiskt kan vara samma sak. En modul kan fungera självständigt och då kallas ett program, men då den slås ihop med en större enhet som består av flera program kan varje del kallas en modul till det större programmet.

Vid en uppdatering av målmiljön då programmet är uppdelat i flera moduler räcker det alltså att bara byta ut de moduler som modifierats. Detta ger att det inte behöver överföras lika mycket data som om hela programmet överförts, med andra ord uppnås en minskning i överföringstid.

[1] Rent strikt är objektorientering en vidareutveckling av moduluppdelning.

För att förklara moduluppdelningstekniken ytterligare följer här ett exempel.

Systemet på målmiljön är uppdelat i tre moduler där den ena innehåller huvudprogrammet, den andra innehåller hjälpfunktioner och den tredje innehåller matematiska funktioner. Om utvecklingen inom matematiken resulterar i att kvadreringsfunktionen kan snabbas upp avsevärt och det är önskvärt att införa denna snabbare funktion på målmiljön kan följande uppdatering ske. Kvadreringsfunktionen ligger i modulen för matematiska funktioner vilket ger att det är denna modul som behöver uppdateras. Vid uppdateringen är det endast nödvändigt att överföra matematikmodulen till målmiljön och byta ut den gamla modulen. Anledningen till att det endast är den modulen som behöver överföras är att de andra två modulerna inte har förändrats någonting vilket leder till att de kan lämnas kvar oförändrade på målmiljön. Efter det att matematikmodulen överförts och satts in i systemet är uppdateringen klar och systemet är redo att tas i bruk.

5.1.1 Krav på målmiljö

Vad som krävs för att kunna utnyttja denna teknik är att målmiljön på något sätt stödjer uppdelning. Denna uppdelning kan till exempel te sig i form av flera filer eller trådar i systemet. Huvudsaken är att de olika delarna kan skiljas åt och att de kan kommunicera med varandra.

5.1.2 Mängden förarbete innan överföring av data kan ske

Detta är beroende på hur utvecklingsmiljön fungerar på värddatorn. Fördelaktigt är givetvis om utvecklingsmiljön har ett bra stöd för moduluppdelning. Om detta saknas, eller är otillräckligt måste mer manuellt arbete ske innan överföring kan ske till målmiljön. Den nya modulen måste först tas fram och identifieras och först därefter kan den nya modulen överföras till målmiljön.

5.1.3 Mängden data som behöver överföras

Om programmet är uppdelat i tillräckligt många moduler så att varje modul blir mycket liten ges en uppenbar fördel. Vid uppdatering skall endast de modifierade modulerna överföras, om dessa moduler är mycket små så resulterar detta i att det inte blir mycket att överföra eftersom endast de moduler som modifierats skall överföras och storleken på dessa är mycket liten. Om däremot programmet består av ett fåtal moduler kan storleken på varje modul bli stor och vid en uppdatering i detta läge kan en stor mängd data behöva överföras trots att moduluppdelning använts.

5.1.4 Mängden efterarbete av data innan start på målmiljön kan ske

Mängden efterarbete kan variera mycket beroende på hur målmiljön är uppbyggd. I de fall ett operativsystem existerar kan oftast detta utföra det mesta av arbetet vad gäller utbyte av gammal modul till ny. Däremot kan det vara så att operativsystemet har en mycket begränsad funktionalitet. I de fall kan uppdaterings-proceduren bli betydligt krångligare vid moduluppdelning. Om minnesutrymmet på målmiljön är direkt åtkomligt från värddatorn blir uppdateringen enkel. Då kan värddatorn agera operativsystem och utföra bytet av de modifierade modulerna på målmiljön. Om det däremot är så att programmet på målmiljön inte är åtkomligt från värddatorn, eller att programmet inte går att modifiera från värddatorn, blir det mycket svårt att utföra

uppdateringen. I dessa lägen kan en uppdatering till exempel innebära att utbyte av en IC-krets måste genomföras och då är det inte frågan om att överföra något program till en målmiljö utan då handlar det om att programmera IC-kretsar.

Om det inte gör något att målmiljön stannar under tiden uppdatering sker så kan detta vara en mycket enkel och smidig procedur. Eftersom de moduler som överförs är nya och klara att tas i bruk är det bara att byta ut de gamla modulerna och sätta dit de nya istället. Hur lätt det sedan är att byta ut dessa moduler varierar från fall till fall.

5.1.5 Möjlighet att undvika avbrott

Då stopp av målmiljön vill undvikas blir proceduren mer komplicerad. För att kunna undvika avbrott av målmiljön under tiden som uppdatering utförs så måste målmiljön ha stöd för flera program eller moduler i minnet samtidigt. Vidare måste även operativsystemet kunna bearbeta och hantera andra data samtidigt som systemet körs. En möjlig lösning på detta är att använda sig av en miljö som kan hantera flera processtrådar samtidigt och då starta upp den nya modulen på en ny processtråd och sedan helt enkelt skifta över till den nya när tillfälle ges.

Ett stort problem då avbrott skall undvikas är uppdatering och kopiering av befintliga variabler. Detta problem minskas då moduluppdelning har använts på ett korrekt sätt eftersom andra moduler i systemet endast kan komma åt den uppdaterade modulens funktioner och variabler via det väldefinierade gränssnittet så fås en stark inkapsling som resulterar i att de enda variabler som behöver behandlas är de som finns inuti modulen som skall uppdateras. Det är bland annat detta som gör den starka inkapslingen så värdefull, att vid en uppdatering behöver inte alla variabler föras över till det nya programmet utan det räcker med de som är i den aktuella modulen.

5.1.6 Storleksberoende

Det enda som påverkar är hur stora moduler som används i systemet eftersom det är dessa som måste överföras vid uppdateringen. Då systemet blir så litet att det inte går att dela upp det i moduler utan att det endast blir en modul som utgör hela programmet så blir vinsten lika med noll. Då det gäller så små system är det ingen idé att använda moduluppdelning eftersom det blir samma sak som om ingen teknik använts. Anledningen till detta är att då den uppdaterade modulen överförs så överförs hela programmet.

5.1.7 Sammanfattning av moduluppdelning

I de fall systemet som skall uppdateras blir tillräckligt stort för att delas upp i flera moduler kommer moduluppdelning att förbättra uppdateringstiden därför att om det räcker med att överföra en mindre del av det hela programmet istället för alltihop så har givetvis överföringstiden minskats. Utöver detta tillkommer lite extraarbete på målmiljön eftersom den gamla modulen måste bytas ut. Programmet måste också då bytas ut och om utbytandet måste äga rum på något sätt så är det bra om överföringstiden kan minskas. Vad som bör beaktas är att det inte är användandet av moduluppdelning som ger snabbare uppdateringstid, utan för att moduluppdelning ska vara effektiv i detta avseende krävs att gränssnittet är noga preciserat.

5.2 Patchning

Grundtanken med patchning är att det ofta vid uppdateringar inte är mycket som förändrats från det gamla programmet. Vid en uppdatering kan det till exempel vara frågan om att förändra ett villkor eller att förskjuta ett gränsvärde. Detta leder till att skillnaderna mellan den gamla versionen av programmet och den nya, uppdaterade versionen är få. Om de olika versionerna är så lika borde det vara onödigt att överföra allting igen eftersom den gamla versionen innehåller mycket information som skulle vara värdefull att kunna återanvända. För att kunna uppdatera det gamla programmet måste vetskap finnas om vad som förändrats. För att få reda på vad som modifierats måste det nya och det gamla programmet jämföras. Vid jämförelsen fås de modifieringar som skall göras på det gamla programmet för att erhålla den nya versionen av programmet. De skillnader som framkommer vid jämförelsen av de olika versionerna isoleras och sammanställs i en egen datamängd. Eftersom förändringar kan ha skett på flera ställen i programmet måste även information om var dessa modifieringar skall göras lagras i datamängden. Efter denna jämförelse och sammanställning av datamängden med förändrings-information är den så kallade patchen klar. Det är sedan denna patch (som innehåller modifieringsinformation) som används för att patcha (modifiera) den gamla versionen för att få fram den nya.

Tekniken går alltså ut på att ha ett program som skall uppdateras och en datamängd som säger vad som skall förändras för att få fram det nya, uppdaterade programmet. Efter det att alla förändringar utförts på det gamla programmet är uppdateringen färdig och programmet klart för att tas i bruk.

Det finns dock ett väsentligt problem med att använda patchning vid uppdateringar. Den patch som erhålls vid jämförelsen kan endast appliceras på den version som användes för att ta fram patchen. All information om förändringar är baserad på en specifik version, om förändringarna utförs på en annan version kan vad som helst inträffa eftersom förändringarna utförs blint på de positioner som specificerats. Detta leder till problem då information saknas om vilken version som är på målmiljön som skall uppdateras.

En möjlig lösningsmetod är att alltid behålla en kopia av det senaste programmet på värddatorn. När sedan en uppdatering utförs jämförs det nya programmet med det som redan finns på värddatorn och ur detta fås en patch som går att applicera på målmiljön. Detta leder till nya problem, som att det till exempel kan vara svårt att veta om den versionen som finns på värddatorn är exakt densamma som finns på målmiljön. Detta problem kan, till exempel, lösas genom att ha en god versionshantering och då endast överföra versionsinformation från målmiljön till värddatorn. Ett alternativ till versionshantering kan vara att använda någon av de olika felkontroller som finns (till exempel Cyclic Redundancy Check (Halsall, 1996)) för att verifiera att versionen på värddatorn är den samma som den på målmiljön.

För att förklara patchningstekniken ytterligare följer här ett exempel.

Det gamla programmet som finns på målmiljön är ”03A3BF3CA7763A5B” som skall föreställa den hexadecimala representationen av programmet. Om sedan en uppdatering skall göras som ska resultera i att programmet skall se ut ”03A3**F**F3C**63**763A5B” ger detta följande. En jämförelse av de olika versionerna görs och skillnaderna är på positionerna 4, 8 och 9 (räknat så att den första har position 0 (noll)). Dessa skillnader skall isoleras och information skall läggas till om var skillnaderna förekom.

Position:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Gammalt program:	0	3	A	3	B	F	3	C	A	7	7	6	3	A	5	B
Nytt program:	0	3	A	3	**F**	F	3	C	**6**	**3**	7	6	3	A	5	B
Förändring:	-	-	-	-	x	-	-	-	x	x	-	-	-	-	-	-

Position	Ändras till
4	F
8	6
9	3

Denna lista skrivs sedan om till en rad där positionen står först direkt följt av respektive förändring, alltså i det här fallet ”4F8693”. Detta är den färdiga patchen som behöver överföras till målmiljön för att utföra uppdateringen, eftersom detta är all den information som behövs för att uppdatera det gamla programmet. I detta fallet minskas den data som behöver överföras med 63% (16 tecken till 6 tecken).

Observera att detta exempel är mycket enkelt och vid stora tillämpningar kan framtagandet av patchen bli betydligt mer komplicerat. Vidare kan det vara så att en mer komplicerad algoritm kan användas för att hitta mer avancerade likheter/olikheter mellan de olika versionerna. Den kan till exempel klara av att se förflyttningar av stora delar vilket kan resultera i en kortare representation än att behöva byta ut tecken för tecken. Grundtanken är dock att få fram en liten datamängd som beskriver hur den gamla versionen av programmet skall modifieras för att få fram den nya versionen.

5.2.1 Krav på målmiljö

Då patchen överförts till målmiljön måste den appliceras på det gamla programmet. Målmiljön måste därför kunna utföra andra uppgifter utöver de den är konstruerad för. Vad som menas är att målmiljön kan behöva utföra andra uppgifter än att till exempel flytta en robot eller läsa av en sensor. Till exempel kan målmiljön behöva kunna manipulera det minnesområde som programmet ligger i för att kunna applicera den patch som överförts från värddatorn. Målmiljön måste alltså kunna utföra de förändringar som patchningen kräver i sin egen miljö för att kunna fullfölja uppdateringen, som synes är detta mycket beroende av vilken patchningsalgoritm som använts och hur komplicerad funktionalitet algoritmen kräver. En annan möjlig lösning är att ett operativsystem finns på målmiljön som kan ta hand om patchningen genom att utföra de modifieringar som finns specifierade i patchen.

5.2.2 Mängden förarbete innan överföring av data kan ske

Det är här själva patchen som skall överföras skall tas fram och som tidigare visats kan detta vara en komplicerad process. Värddatorn är dock oftast tillräckligt snabb för att klara av denna uppgift på kortare tid än om överföring skulle skett av hela programmet.

5.2.3 Mängden data som behöver överföras

Tanken med patchning är just att få fram så lite som möjligt att överföra. Detta innebär att det oftast är mycket lite data som behöver överföras till målmiljön vid uppdatering. Vid vissa extrema fall kan patchningstekniken till och med minska den datamängd som skall överföras med 100%. Detta kan ske i de fall där programmeraren tror sig ha löst en algoritm på ett snabbare sätt men det sedan visar sig att kompilatorn redan har löst algoritmen på detta sätt. I så fall blir det ingen skillnad på den nya versionen från den gamla och detta leder till att det inte finns någon anledning att överföra någonting till målmiljön eftersom ingen förändring har skett som kommer att påverka målmiljön.

5.2.4 Mängden efterarbete av data innan start på målmiljön kan ske

Eftersom patchen måste appliceras på målmiljön kan det bli en del arbete att utföra innan start på målmiljön kan ske. Mängden arbete är beroende av hur komplicerad den algoritm är som skall användas för att få fram vad som skall ändras. Om algoritmen är enkel blir det lite att göra på målmiljön (kanske en hel del data som måste behandlas, men med mycket enkla operationer) och om algoritmen är komplicerad kan det bli mer arbete att göra.

5.2.5 Möjlighet att undvika avbrott

Då en patch appliceras modifieras programmet direkt på bit-nivå. Detta innebär att om programmet körs samtidigt kan det bli fel om läsning skulle ske innan hela patchen applicerats. Av detta följer att antingen så måste målmiljön stödja att filen med programmet modifieras under tiden som kopian av programmet i minnet körs, eller också måste en kopia göras på målmiljön så att patchen kan appliceras på kopian som inte körs eller används för tillfället. Då det nya programmet skall börja köra istället för det gamla uppstår komplikationer. Alla variabler som finns i minnet måste lagras för att det nya programmet skall kunna ta del av dem, vilket leder till att mycket information måste kopieras i minnet. Det blir mer komplicerat om variabler byter namn vid uppdateringen.

Att försöka undvika avbrott vid större uppdateringar är mycket svårt eftersom många problemställningar uppstår (till exempel kan modifieringen leda till att en snurrande trumma på 2 ton skall byta håll, eller också har ett antal variabler bytt namn samtidigt som andra variabler har tagits bort).

5.2.6 Storleksberoende

Eftersom patchningen arbetar på bit-nivå är den oberoende av vilken typ av system den uppdaterar, det kan vara ett mycket litet program eller ett stort system. En nackdel är då det är frågan om riktigt stora uppdateringar för då kan det bli så att patchen blir

lika stor, om inte större, än hela systemet var från början. I sådana fall blir det givetvis onödigt att överföra en patch till målmiljön eftersom denna information bara resulterar i onödigt extra-arbete då alla positioner måste läsas och målsystemet måste ändras lite i taget. I det läget hade det varit betydligt smidigare att överföra hela programmet och bytt allt på en gång.

Beakta exemplet i kapitel 5.2. Om vartenda tecken hade förändrats i den nya versionen med den teknik som visats här hade patchen blivit dubbelt så stor som originalet. Just i detta specialfall hade en förlust erhållits på 100%, vilket är helt oacceptabelt.

5.2.7 Sammanfattning av patchning

Eftersom patchning är konstruerad just för att få fram de små förändringar som skett är det en ypperlig teknik att använda vid uppdateringar då mindre modifieringar i systemet skett. Som framkommit under analysen så är inte patchning det bästa alternativet då det gäller stora uppdateringar men vid mindre är det en mycket bra teknik. En nackdel med patchning är att det kan bli mycket svårt att undvika att stanna målmiljön under tiden som uppdatering sker. Denna rapport tar inte upp kombinationer av olika överföringstekniker men detta är ett typiskt fall där en kombination vore mycket fördelaktig. Om patchningstekniken endast användes då det gällde riktigt små modifieringar och att vid större modifieringar använda till exempel komprimering eller moduluppdelning.

5.3 Komprimering

Komprimering är en teknik som idag används i väldigt stor omfattning då det gäller att överföra datamängder mellan datorer. De komprimeringsalgoritmer som används kan vara väldigt komplicerade och på så vis krävande av det system de körs på, men eftersom datorer nu för tiden är bättre på att beräkna än att överföra data så används ändå de tillgängliga resurserna på ett bättre sätt. Tanken är alltså att utnyttja ren processorkraft under en tid för att sedan få en mindre datamängd att överföra till målmiljön. Efter det att den komprimerade datamängden överförts till målmiljön måste den dekomprimeras för att kunna användas. Dekomprimeringen är givetvis också processorkrävande men, som snart skall visas, så är det ofta så att dekomprimering är enklare att utföra än komprimering. Minskningen av datamängden som skall överföras ska helst åtminstone motsvara den tid det tog för komprimeringsalgoritmen att få fram vad som skulle överföras sammanslaget med den tid det tar att dekomprimera datamängden på målmiljön. Om till exempel en överföringshastighet på 56 kbits/sekund skall användas så innebär det att 35 kilobyte kan överföras på 5 sekunder. Alltså har processorn cirka 2,5 (antag att det tar lika lång tid att komprimera som att dekomprimera) sekunder på sig att minska datamängden som skall överföras med minst 35 kilobyte. Om minskningen blir större ges en vinst i form av snabbare överföring men om minskningen däremot blir mindre förloras tid på så sätt att den totala överföringstiden ökar. Observera att även om det blir mindre data att överföra så behöver det inte innebära att den totala uppdateringstiden förkortas eftersom det även tar tid att utföra komprimering och dekomprimering.

En del komprimeringsalgoritmer (till exempel Huffman (Lynch, 1985), Adaptive Arithmetic (Salomon, 1998)) baseras på att först ta fram en tabell eller någon form av

lista. Denna lista innehåller olika saker beroende på algoritm men vanligt är att listan innehåller sannolikheter för förekomster av olika tecken i den aktuella datamängden. Vid denna typ av algoritm måste listan som genererats överföras till målmiljön för att dekomprimering skall gå att genomföra. Andra typer av komprimeringsalgoritmer använder istället ett begränsat minne till att komma ihåg vad som nyligen komprimerats. Detta minne av de senaste komprimerade tecknen används till att komprimera nästkommande tecken. Det finns många algoritmer som använder denna teknik och exakt vad som görs är väldigt olika. Vanligtvis så utförs en sökning i det avsatta minnesutrymmet för att hitta likheter mellan det som ska läggas till och det som redan komprimerats (till exempel LZ77, Front Compression (Salomon, 1998)).

Komprimeringsalgoritmer som baseras på principen att det som inte märks kan tas bort används oftast till att komprimera bilder, ljud och video (till exempel JPEG (Salomon, 1998)). Dessa komprimeringsalgoritmer använder den enklaste av alla komprimeringsprinciper, ta bort (eng. lossy compression (Salomon, 1998)). Komprimeringsalgoritmer av den här sorten lämpar sig utmärkt till bilder, ljud och video eftersom den komprimerade datamängden skall tolkas av människor som ändå inte kan se någon skillnad mellan den komprimerade och okomprimerade datamängden även om dessa två egentligen inte är likadana. Då dessa algoritmer baseras på att ta bort information lämpar de sig naturligtvis inte alls för att komprimera program eftersom det kan få väldigt radikala följder om information förloras vid komprimeringen. Av detta följer att komprimeringsalgoritmer som tar bort information ur den datamängd som skall komprimeras utesluts.

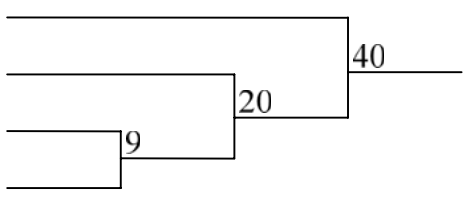
Som visats tidigare så innebär komprimering ofta att hitta likheter, upprepningar eller någon form av symmetri i den datamängd som skall komprimeras. För att dessa egenskaper skall hittas utförs oftast någon form av sökning eller jämförelse under tiden som komprimeringen fortskrider. Vid dekomprimering har dessa egenskaper identifierats och den komprimerade datamängden innehåller information om hur datamängden skall dekomprimeras. Av detta följer att dekomprimering blir mycket enklare att genomföra eftersom det bara är att utföra de kommandon som finns lagrade i datamängden. Kommandon som kan finnas lagrade kan till exempel innebära att tecknet som följer skall upprepas 5 gånger. Dessa kommandon måste identifieras vid komprimeringen varför den är mer krävande än dekomprimering. Enligt Salomon (1998) är det oftast mycket komplext att förbättra en komprimerings-algoritm och en vinst på 1% brukar medföra en ungefärlig ökning i beräknings-komplexitet med 10%.

Komprimeringstekniken skall nu belysas genom att komprimera det fiktiva programmet ”13AAAA3A6A13A36A361A33AA3A13A63AAA3A6AAA” med hjälp av huffman-komprimering.

Huffman-komprimering fungerar så att först tas en lista fram på vilken sannolikhet de olika tecknen har att förekomma i det aktuella programmet. Med andra ord räknas antalet förekomster av varje tecken i programmet och detta antal divideras sedan med det totala antalet tecken i programmet. Tag sedan och bind ihop de två tecknen med lägst sannolikhet och beräkna den sammanslagna sannolikheten för dessa två. Upprepa detta tills den totala sannolikheten är ett. Nedan har endast antalet

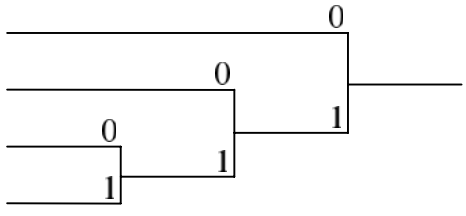
förekomster räknats, för att få fram sannolikheten skall varje tal i detta exempel divideras med 40 (det totala antalet tecken i programmet).

Tecken	Antal
A	20
3	11
6	5
1	4



När detta är klart har ett träd som visas ovan byggts upp. Sedan tilldelas varje förgrening en etta eller en nolla. Den övre förgreningen skall ha en nolla och den lägre en etta. Efter det att dessa ettor och nollor är placerade är trädet färdigt (se nedan). För att få fram de binärkoder som de olika tecknen i programmet skall ha traverseras trädet från höger till vänster tills aktuellt tecken har hittats. Det är detta träd som används till att komprimera och dekomprimera datamängden som skall skickas mellan värddatorn och målmiljön.

Tecken	Binärkod
A	0
3	10
6	110
1	111



Om till exempel sekvensen ”A3A” skall överföras så skickas följande bitar ”0100”. På den mottagande sidan traverseras trädet från höger till vänster tills ett tecken har kommit ut, därefter börjar traverseringen om från höger med nästa tecken. Detta repeteras tills överföringen är klar. Först in är ”0” vilket leder direkt till tecknet ”A”. Därefter kommer ”1” vilket gör att traverseringen väljer den nedre delen av den första förgreningen. Sedan kommer ”0” vilket leder till att den övre förgreningen väljs vid den andra förgreningen och det leder till tecknet ”3”. Därefter kommer ”0” vilket ger ett ”A” till.

För att kunna dekomprimera datamängden på den mottagande sidan måste alltså mer information över än bara den komprimerade datamängden. Till exempel måste sannolikheterna för de olika tecknen skickas med eller också kan hela trädet skickas men exakt vad som krävs är beroende på vilken typ av komprimeringsalgoritm som används.

Detta exempel belyser på ett klart sätt att det kan vara mycket mer komplicerat att komprimera en datamängd än det kan vara att dekomprimera densamma.

5.3.1 Krav på målmiljö

Eftersom komprimeringsalgoritmer kan vara mycket krävande så krävs i alla fall en tillräckligt snabb processor som inom rimlig tid klarar av att dekomprimera den datamängd som överförts. Med tillräckligt snabb menas att de tidskrav som finns för det specifika systemet inte överskrids.

5.3.2 Mängden förarbete innan överföring av data kan ske

Vitsen med att använda komprimering är att utnyttja tillgängliga resurser på ett bättre sätt. Processorn är ofta en resurs som inte används fullt ut vid överföring och det är detta komprimeringstekniken utnyttjar. Det är mycket arbete som utförs på värddatorn innan överföringen kan börja men då detta arbete till största delen är beräkningar och jämförelser så klarar en dator detta snabbt. Det är under denna fas som den komprimerade datamängden som skall överföras tas fram.

5.3.3 Mängden data som behöver överföras

Mängden data som behöver överföras är beroende på vilken komprimeringsalgoritm som används och hur avancerad denna är. En mer avancerad algoritm kanske kan minska datamängden mer än en mindre avancerad men den tar också mer resurser till sitt förfogande. En vinst på 50% brukar anses som normalt då programfiler komprimeras och det innebär en halvering av överföringstiden (Lynch, 1985; Williams, 1991).

5.3.4 Mängden efterarbete av data innan start på målmiljön kan ske

Vad som behöver utföras på målmiljön är att dekomprimera den datamängd som överförts. Som visats tidigare är dekomprimering en mindre komplicerad process än komprimering men om tidskrav föreligger som till exempel begränsar tiden för en omstart så måste detta beaktas noga.

5.3.5 Möjlighet att undvika avbrott

Den datamängd som överförts innehåller hela det system som skall köras på målmiljön. Detta innebär att om avbrott skall undvikas vid uppdatering så måste alla variabler lagras för att kunna överföras till den nya versionen. Att undvika avbrott då hela systemet komprimerats och överförts på en gång väcker samma problem som om systemet överförts utan komprimering. Komprimering påverkar varken positivt eller negativt då det gäller att undvika avbrott på målmiljön.

5.3.6 Storleksberoende

Storleksberoendet för komprimering är mycket beroende av vilken typ av komprimeringsalgoritm som använts. Då komprimeringsalgoritmer ofta letar efter likheter och mönster i den datamängd som skall komprimeras är en större datamängd bättre. Anledningen är att om datamängden ökar så ökar också chansen för att nya mönster eller likheter framträder. Av samma anledning kan det vara svårt att hitta likheter och mönster i en liten datamängd, det finns inte så mycket data att vara likheter mellan. Vid små datamängder bör enklare komprimeringsalgoritmer användas för att undvika för mycket overhead i form av tabeller och översättningslistor. Exempel på komprimeringsalgoritmer som bör undvikas vid små datamängder är de som först genererar någon form av sannolikhetstabell över de förekommande tecknen i datamängden. Om datamängden är liten så kommer tabellen att vara stor i jämförelse, vilket till och med kan leda till att det som skall överföras blir större än den ursprungliga datamängden eftersom denna tabell också måste överföras till målmiljön för att dekomprimering skall gå att genomföra.

5.3.7 Sammanfattning av komprimering

Komprimering är en bra teknik att använda oavsett om det gäller stora eller små system. Vad som bör beaktas är att vinsten ofta ligger kring 50 % och om detta inte är tillräckligt så bör någon annan teknik användas. Komprimering utnyttjar beräkningskapaciteten på värddatorn för att minska datamängden att överföra till målmiljön. Om värddatorn inte kan belastas av detta eller om målmiljön inte kan nå upp till de krav som komprimeringsalgoritmerna ställer så bör andra överförings-tekniker användas. En mycket bra sak med komprimering är att det går att applicera på i princip vilken typ av system som helst. Vid små system (räknat i bytes), eller där beräkningskapaciteten inte är speciellt stor, bör en enklare komprimeringsalgoritm användas men om det är så att värddatorn är snabb på att beräkna och ändå inte används till något annat så kan en mer avancerad komprimeringsalgoritm vara fördelaktigt.

5.4 Sammanfattning och slutsatser av jämförande studie

Då en snabb och enkel lösning skall uppnås är komprimering av data en ypperligt bra teknik. Även om det innebär en halvering av överföringstiden så kan detta vara värdefullt eftersom det är en mycket enkel teknik att införa i de flesta system. Om önskemål däremot föreligger om att minska överföringstiden ytterligare bör tanke ges åt att det kan innebära mycket mer komplexa beräkningar om en liten ökning skall uppnås med hjälp av komprimering.

Om en mer långsiktig lösning skall uppnås är det bättre att satsa på att införa moduluppdelning och göra det på ett korrekt sätt. Detta är en mycket bra teknik att använda, inte bara ur överföringssynpunkt utan den har även många andra fördelar så som att det till exempel blir enklare att undvika avbrott på målmiljön. Som nämnts räcker det inte med att bara dela upp programmet i flera moduler och sedan låta det sköta sig självt utan arbete måste läggas ner för att få moduluppdelning att fungera som det är tänkt. Då moduluppdelning delar upp programmet i flera mindre bitar ger detta en uppenbar fördel då endast den lilla del som modifierats behöver överföras vid uppdatering.

Dock så är patchningstekniken en klar vinnare då det gäller att utföra mindre modifieringar på målmiljön. Något som kan spela en stor roll här är att denna teknik just är specialiserad på mindre modifieringar. På grund av detta så är patchnings-tekniken olämplig, gentemot de andra överföringsteknikerna här, då det gäller mycket stora modifieringar och vid ett totalt byte av kompletta system.

6 Praktisk applicering

De resultat som framkommit under litteraturstudien har applicerats på ett praktiskt exempel. Det system som undersöktes är BiFas UHS (se Bilaga 1 (Binär, 1999)).

De anslutningsmöjligheter som finns att tillgå är i standardutförandet en serie-anslutning. Hastigheten som används idag på denna port är 38.400 bit/sekund och det protokoll som används vid filöverföring är Zmodem. Utöver serieanslutningen är det även möjligt att ansluta en separat enhet där ett extra alternativ ges i form av en ethernetanslutning. Den extra enheten ger då möjlighet att ansluta via tvinnat trådpar (TP). Hastigheten vid denna anslutning är 10 Mbit/sekund och här används TCP/IP som protokoll vid filöverföring.

Alla tester är utförda med RS-232-interfacet som är sammankopplat med värddatorn via en seriekabel. Under den praktiska delen av arbetet användes en dator som var bestyckad med en Intel 166 MHz processor och 32 Mbyte RAM-minne. Denna dator fick agera värddator under testerna. Specifikationer på målmiljön återfinns i Bilaga 1.

Utvecklingsmiljön för BiFas UHS fungerar så att det program som gjorts först översätts till C-kod och därefter genereras ett dynamiskt länkbibliotek (eng. dynamic link library (dll-fil)). Det är sedan denna dll-fil som överförs till BiFas UHS och där kommer att utgöra det nya programmet. Ett färdigutvecklat och kompilerat program har ungefär storleken 1 Mbyte vilket medför att överföringen är en flaskhals och tar upp mycket tid då serieanslutningen används.

Först kommer en teoretisk analys att genomföras där resultaten från den jämförande studien kommer att ligga till grund för beslut om vilka tekniker som kan komma på tal att användas. I denna analys kommer även eventuella anledningar till att inte använda en viss teknik att framträda. Därefter presenteras resultaten från appliceringarna av de olika överföringsteknikerna på BiFas UHS. Slutligen sammanställs slutsatser för den praktiska delen av arbetet.

6.1 Analys

Detta kapitel kommer att undersöka huruvida de olika överföringsteknikerna lämpar sig för den utvecklingsmiljö som används för BiFas UHS idag.

6.1.1 Moduluppdelning

På grund av den uppbyggnad som används idag i utvecklingsmiljön är inte moduluppdelning möjlig. Det program som genereras kompileras till en enda dll-fil som sedan skall överföras och det finns inget stöd idag för att generera fram specifika delar av systemet. Målmiljön BiFas UHS har däremot inga problem med att hantera moduluppdelning eftersom denna har ett filsystem som kan hantera de olika modulerna. Vid en eventuell vidareutveckling av utvecklingsmiljön skulle modifieringar kunna ske så att moduluppdelning kan bli aktuellt. Begränsningarna ligger alltså på värddatorn i utvecklingsmiljön, inte i målmiljön.

6.1.2 Patchning

För att patchning skall vara givande att använda måste ett redan befintligt system finnas på BiFas UHS eftersom patchningstekniken bygger på uppdatering. Vanliga

modifieringar som utförs då snabb uppdatering önskas är att lägga till eller ta bort enstaka kontakter (brytare) och att modifiera variabler (Binär, 1999). Variabel-modifieringar som görs kan till exempel innebära att byta absolutadress för variabeln i minnet på målmiljön.

I dagsläget är det möjligt att modifiera proceduren mellan generering av dll-fil och överföring. Utöver detta är det även möjligt att modifiera startproceduren på målmiljön vilket tillsammans leder till att patchning är möjlig.

Valet av program att utföra patchning med blev enkelt. Det enda program som fanns tillgängligt för detta arbete var RTPatch från PocketSoft (PocketSoft, 1999).

6.1.3 Komprimering

I dagsläget är det möjligt att modifiera proceduren mellan generering av dll-fil och överföring. Utöver detta är det även möjligt att modifiera startproceduren på målmiljön vilket tillsammans leder till att komprimering skulle kunna vara möjlig. Salomon (1998) nämner två komprimeringsprogram som skulle kunna lämpa sig för den här typen av system som kör operativsystemet MS-DOS. Det ena är PKlite från PKWare (PKWare, 1999) och det andra är DIET från Teddy Matsumoto (Matsumoto, 1999). Båda dessa program är så kallade EXE-komprimerare, vilket innebär att filerna som komprimeras inte behöver dekomprimeras manuellt utan de dekomprimeras direkt till minnet varje gång de skall användas. Valet bland dessa två föll på DIET från Teddy Matsumoto. Den avgörande anledningen är att PKlite inte kan komprimera de dll-filer som skall överföras till BiFas UHS, vilket DIET kan.

DIET fungerar så att vid komprimering av datafiler så måste ett TSR-program (Terminate and Stay Resident) köras på målmiljön och detta program känner av om en komprimerad datafil skall läsas. Vid läsning av en komprimerad datafil känner DIET av detta och dekomprimerar filen utan att kringliggande program märker något.

6.2 Resultat av applicering

Det system som skall överföras är en dll-fil som omnämnts tidigare i kapitel 6. Denna dll-fil utgör hela det system som måste finnas på målmiljön för att allting ska fungera.

Till dessa praktiska tester har fem olika system använts. Eftersom ett system endast innehåller en dll-fil är det även möjligt att se dessa som fem olika program. De program som använts är utvecklade av Binär och källkod finns tillgänglig till fyra av dessa. Det är till det största programmet, räknat i bytes, där källkod inte finns tillgängligt. De två minsta programmen är demonstrationsprogram och är således mycket enkelt uppbyggda. Däremot är de större programmen mer komplexa och innehåller en hel del programkod.

De tider som visas är uppmätta med det överföringsprogram som används i utvecklingsmiljön på värddatorn.

I tabell 1 och diagram 1 visas hur uppdatering av systemet sker om ingen speciell överföringsteknik används. Detta är den tid det tar i dagsläget att uppdatera ett BiFas UHS system.

Storleken på det kompletta system som skall överföras	Tid för överföring
73 728 bytes (1)	22 sek
86 016 bytes (2)	27 sek
147 456 bytes (3)	45 sek
262 144 bytes (4)	1 min, 18 sek
1 077 248 bytes (5)	5 min, 21 sek

Tabell 1

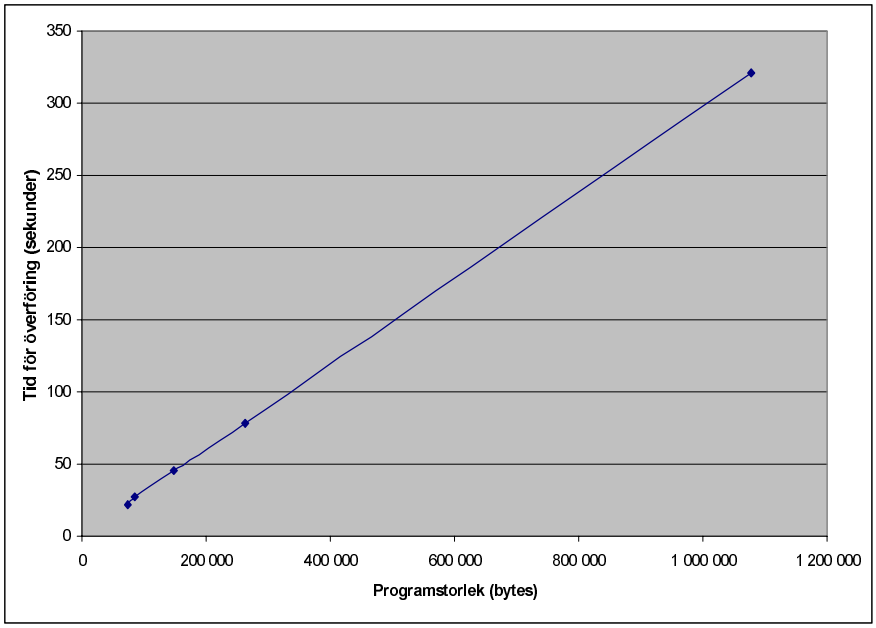


Diagram 1

6.2.1 Komprimering

Den första tekniken som testas är komprimering. Den storlek som visas i tabell 2 och diagram 2 är efter komprimering och tiden är den tid det tar att överföra respektive system till BiFas UHS.

Storleken efter förarbete på det kompletta system som skall överföras	Tid för överföring
28 091 bytes (1)	9 sek
34 051 bytes (2)	11 sek
56 215 bytes (3)	18 sek
101 826 bytes (4)	31 sek
415 196 bytes (5)	2 min, 3 sek

Tabell 2

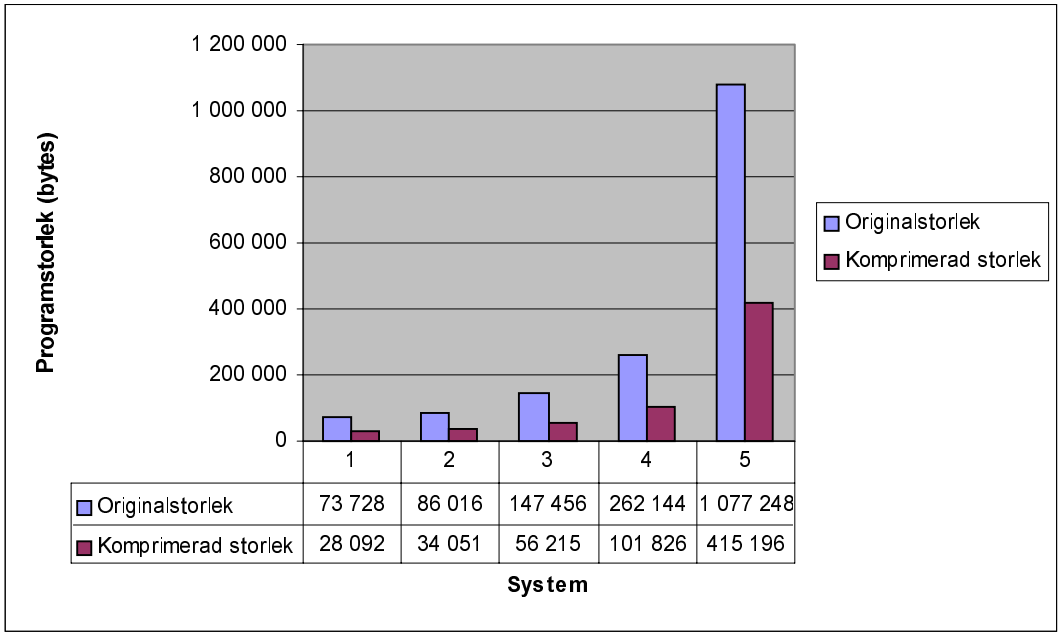


Diagram 2

Diagram 3 visar att minskningen är lika stor, procentuellt sett, oavsett om ett stort program eller ett litet program överförs. Programstorleken minskas till cirka 40% av den ursprungliga storleken, vilket leder till en minskning av överföringstiden med 60%.

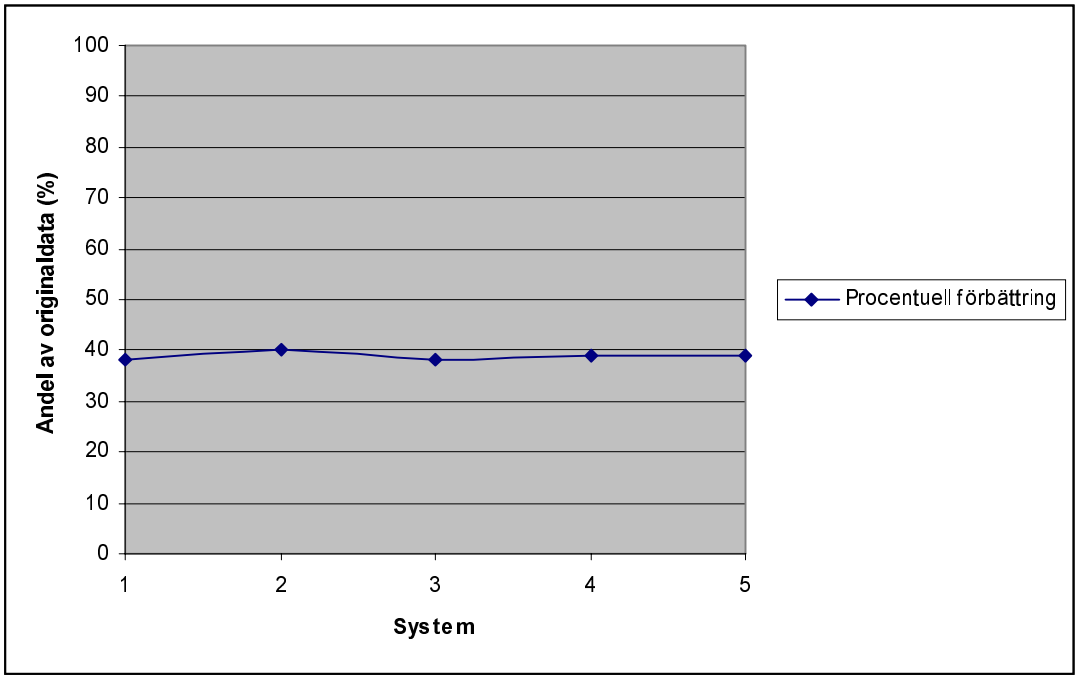


Diagram 3

Som påpekats tidigare så är det inte bara överföringstiden som är väsentlig vid uppdatering. Tiden det tog att komprimera dessa system har givetvis också beaktats. På värddatorn tog komprimeringen cirka 1 sekund i alla fall utom för det allra största systemet, vilket tog cirka 3 sekunder. Utöver detta tillkommer även den tid det tar att

dekomprimera systemet på målmiljön. Direkt efter det att programmet har överförts till BiFas UHS görs en omstart av systemet och en kontroll görs huruvida programmet startades om korrekt eller ej. För varje fall utom det största var det inga problem vid omstart, målmiljön startade om precis som vanligt. Omstart av det största programmet tog längre tid än vad de andra tog, vilket medförde att programmet som kontrollerade att målmiljön startat upp korrekt inte fick något svar direkt efter överföringen av det uppdaterade programmet. Programmet drog då slutsatsen att målmiljön inte fungerade som den skulle och rapporterade detta. Dock visade det sig vid en manuell kontroll att programmet körde som väntat på målmiljön.

6.2.2 Patchning

Som tidigare nämnts är patchning endast givande vid små uppdateringar. Detta medför att modifieringar måste göras till de olika systemen och där se hur mycket som behöver överföras för att uppdatera målmiljön. Då källkod endast finns tillgänglig till de första fyra systemen så kommer patchningstekniken att endast testas på dessa fyra vid olika, vanliga modifieringar. Utförda modifieringar beskrivs senare i kapitlet. De modifieringar som applicerats på de olika systemen innebar inte någon storleksförändring för programmen utan bara modifieringar i den inre strukturen.

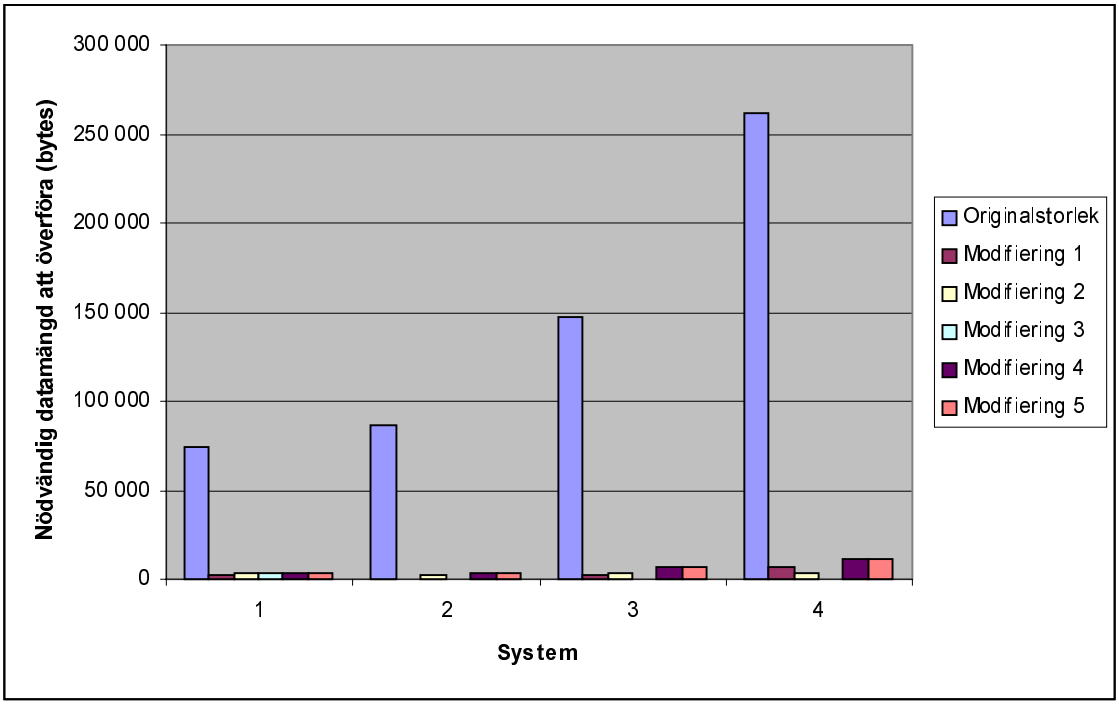


Diagram 4

Den patchstorlek som visas i diagram 4 och tabell 3 är den datamängd (räknad i bytes) som behöver överföras till målmiljön för att utföra en komplett uppdatering. Som synes är denna betydligt mindre än programmets originalstorlek vilket leder till en stor minskning av den datamängd som behöver överföras och således en stor minskning av överföringstiden. Efter patchstorleken i tabell 3 står utvecklingstiden för respektive patch, vilket är den tid det tog för patchprogrammet att hitta vad som förändrats i de olika programmen.

Modifiering 1: Ändra initialt variabelvärde för en variabel från 0 till 1001 (decimalt)

System	Patchstorlek	Utvecklingstid för patch	Beräknad överföringstid
73 728 (1)	2 778	<1 sekund	1 sekund
86 016 (2)	223	<1 sekund	0,5 sekunder
147 456 (3)	2 122	2 sekunder	0,5 sekunder
262 144 (4)	6 771	2 sekunder	2 sekunder

Modifiering 2: Lägga till en global variabel med värdet 1001 (decimalt)

System	Patchstorlek	Utvecklingstid för patch	Beräknad överföringstid
73 728 (1)	3 259	<1 sekund	1 sekund
86 016 (2)	2 698	<1 sekund	1 sekund
147 456 (3)	3 304	2 sekunder	1 sekund
262 144 (4)	3 515	2 sekunder	1 sekund

Modifiering 3: Byta absolutadress för en variabel till 101 (hexadecimalt)

System	Patchstorlek	Utvecklingstid för patch	Beräknad överföringstid
73 728 (1)	3 271	<1 sekund	1 sekund
86 016 (2)	189	<1 sekund	0,5 sekunder
147 456 (3)	221	2 sekunder	0,5 sekunder
262 144 (4)	187	2 sekunder	0,5 sekunder

Modifiering 4: Lägga till en kontakt (brytare)

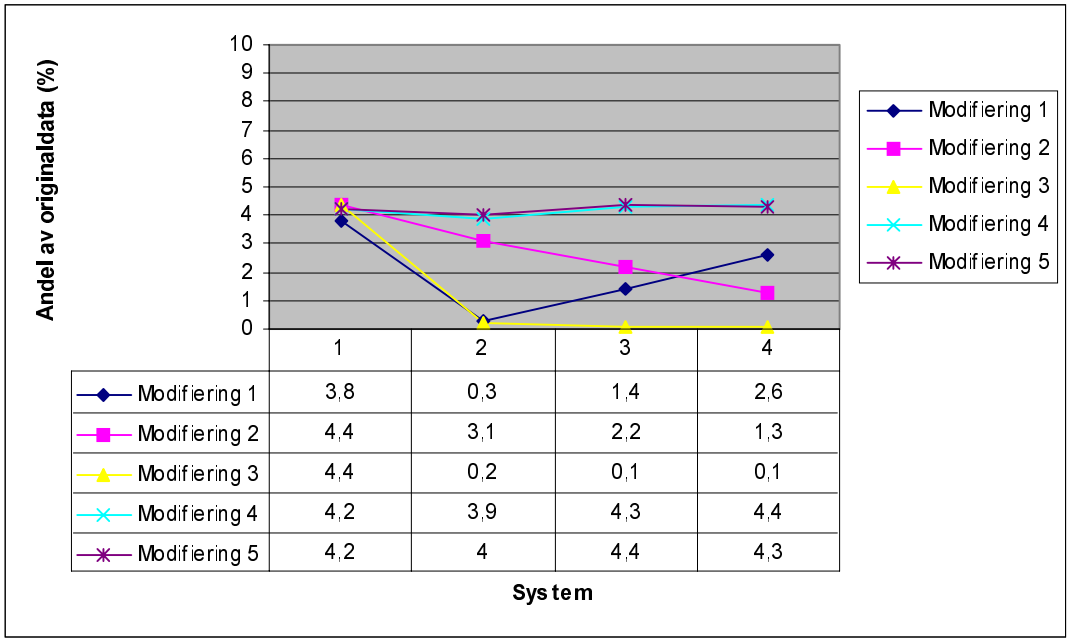
System	Patchstorlek	Utvecklingstid för patch	Beräknad överföringstid
73 728 (1)	3 121	<1 sekund	1 sekund
86 016 (2)	3 349	<1 sekund	1 sekund
147 456 (3)	6 271	2 sekunder	2 sekunder
262 144 (4)	11 546	2 sekunder	3,5 sekunder

Modifiering 5: Ta bort en kontakt (brytare)

System	Patchstorlek	Utvecklingstid för patch	Beräknad överföringstid
73 728 (1)	3 115	<1 sekund	1 sekund
86 016 (2)	3 463	<1 sekund	1 sekund
147 456 (3)	6 439	2 sekunder	2 sekunder
262 144 (4)	11 174	2 sekunder	3,5 sekunder

Tabell 3

I tabell 3 syns två klart avvikande värden, modifiering 1 – system 2 och modifiering 3 – system 1. Den första kan bero på att den variabel som modifierades hade en absolut minnesadress och detta medför att en fixerad storlek avsatts för denna variabel då variabeltypen är känd och var i minnet den skall ligga. Den andra avvikelsen kan bero på att den variabel vars adress förändrades inte hade någon fixerad minnesadress innan modifieringen. Detta leder till att extra information läggs till i programmet och medför en större förändring än om adressen bara hade bytts ut.



	1	2	3	4
Modifiering 1	3,8	0,3	1,4	2,6
Modifiering 2	4,4	3,1	2,2	1,3
Modifiering 3	4,4	0,2	0,1	0,1
Modifiering 4	4,2	3,9	4,3	4,4
Modifiering 5	4,2	4	4,4	4,3

Diagram 5

Diagram 5 visar att den procentuella minskningen är jämn och för mindre modifieringar av målmiljön ligger storleken på patchen på cirka 5% av originaldatans storlek. Då endast 5% av originaldatan skall överföras ges uppenbarligen en stor vinst i form av förkortad överföringstid till målmiljön.

Då patchningsprogrammet som använts var en utvärderingsversion var det tvunget att trycka på en knapp för att starta appliceringen av patchen på det gamla systemet som skulle uppdateras. Då BiFas UHS saknar tangentbord blev det en omöjlighet att utföra en praktisk applicering på målmiljön. Mätning av tidsåtgång på värddatorn ger åtminstone information om det är för krävande att applicera dessa patcher på målmiljön. Appliceringstiden för de olika systemen blev under 1 sekund för de första 3 systemen och cirka 1 sekund för det största (4). Detta visar på att det fortfarande kan vara möjligt att applicera dessa patchningar på målmiljön utan att det tar för lång tid.

6.3 Slutsatser från praktiskt exempel

Komprimeringstekniken var mycket enkel att införa i det system som användes för detta arbete. Detta tillsammans med att tekniken gav en vinst på nästan 60% gör att den är ett bra val om en enkel lösning på problemet skall uppnås och en större vinst inte behövs. Dock ger lösningen inte en ögonblicklig uppdatering utan det tar fortfarande en del tid att utföra denna, men om tidsvinsten är tillräcklig är detta en

mycket enkel lösning på uppdateringsproblemet för det system som undersökts i detta arbete.

Patchningstekniken bör tas i beaktning om större tidsvinster är nödvändiga, eller då det mest är frågan om mindre uppdateringar. Det innebär lite mer arbete att införa denna teknik i systemet men den kan i gengäld ge en oerhört stor vinst vid mindre uppdateringar. Då det är vid slutet av utvecklingen som det sker många små modifieringar för att få systemet att passa in i den miljö den skall arbeta med, följer att den överföringsteknik som passar bäst är patchning.

7 Slutdiskussion

7.1 Slutsatser

I detta arbete har en studie gjorts huruvida uppdateringstiden för en målmiljö kan förkortas med hjälp av olika överföringstekniker. De överföringstekniker som undersökts är moduluppdelning, patchning och komprimering. Av dessa är komprimering den mest allsidiga lösningen då den är enkel att applicera på många system. Komprimeringstekniken kan användas för både stora och små system och en vinst i form av en halvering av uppdateringstiden uppnås. Komprimeringstekniken ser till att utnyttja den ofta lediga resursen vid överföringar, processorn. Med hjälp av olika avancerade algoritmer minskas den datamängd som skall överföras och därmed fås en vinst i form av en minskning av den datamängd som behöver överföras till målmiljön. Till överföringstiden tillkommer också den tid det tar att komprimera och dekomprimera programmet som skall överföras till målmiljön. Då de datorer som finns idag är så pass kraftfulla att de ofta klarar av att utföra komprimering och dekomprimering inom några få sekunder innebär detta att den extra tid som tillkommer är relativt liten i förhållande till den tid det tar att överföra datamängden. Eftersom de extra tider som tillkommer är förhållandevis små blir resultatet att den totala uppdateringstiden minskar.

I de fall då uppdateringen behöver gå ännu fortare kan de andra överföringsteknikerna beaktas. Patchning är den klart snabbaste tekniken att uppdatera med då det gäller mindre modifieringar i systemet. Patchningstekniken använder sig av en teknik som gör att den blir oslagbar vid mindre modifieringar. Vinsten vid mindre modifieringar med hjälp av patchning kan ligga så högt som 99%. En annan fördel med patchningstekniken är att den automatiskt upptäcker om uppdatering verkligen är nödvändigt vid varje försök att uppdatera målmiljön. I de fall då ingenting är förändrat från den version som redan finns på målmiljön ser patchningen detta och ser då till att inte skapa någon patchningsfil. Av detta följer att då det inte finns något att föra över så är det inget som förs över, alltså kan onödiga överföringar undvikas i dessa fall.

Moduluppdelning är en teknik som liksom patchning och komprimering kan ge en god vinst vid uppdatering av en målmiljö om tekniken appliceras på ett korrekt sätt. Nackdelen med denna teknik gentemot de övriga överföringsteknikerna är att kraven på både värddator och målmiljön ökar. Däremot kan moduluppdelningen medföra en vinst på så sätt att det blir lite att överföra till målmiljön, vilket leder till att en stor minskning av uppdateringstiden uppnås. Utöver den uppenbara fördelen att det blir lite att överföra till målmiljön medför även moduluppdelning flera andra fördelar. Till exempel kan moduluppdelning (och användandet av lös koppling) underlätta felsökning då det oftast är fel i en specifik modul. Det blir alltså inte nödvändigt att gå igenom hela programmet för att hitta ett fel.

De överföringstekniker som kan ses som enklast att introducera i den miljö som används är patchning och komprimering. Däremot kan det bli mer komplicerat att införa moduluppdelning i motsvarande system. Den största anledningen till att det är så stor skillnad på de tre teknikerna är att både patchning och komprimering arbetar på bitnivå och är således helt oberoende av vad de behandlar. Moduluppdelning

opererar däremot på en högre nivå än vad de andra två gör. Eftersom modul-uppdelning arbetar mer på programnivå blir samspelet med målmiljön mycket mer påtagligt. Då detta samspel ökar måste även samspelet med värddatorn öka eftersom det är därifrån som det uppdaterade programmet kommer. Av detta följer att moduluppdelning är mycket mer beroende av vilken miljö som används än vad patchning och komprimering är.

7.2 Framtida arbete

Vad som legat utanför detta arbete är eventuella kombinationer av de olika överföringsteknikerna. Vad som är intressant är att, till exempel använda patchning vid mindre modifieringar men fortfarande ha stöd på värddator och målmiljö för att kunna skicka komprimerad data, eller nya moduler. En möjlig kombination av alla dessa tre tekniker samtidigt skulle kunna vara att *komprimera en patch till en modul.* Detta skulle använda alla tre teknikerna på samma gång och huruvida detta resulterar i en minskad uppdateringstid får en framtida undersökning utvisa.

Detta arbete har inte utfört några praktiska tester med moduluppdelning då den miljö som använts inte klarade av detta. Vid en eventuell vidareutveckling av mjukvaran på värddatorn bör emellertid beaktning tas åt att moduluppdelning skulle kunna lösa fler problem än de som tagits upp i denna rapport. Till exempel skulle det vara möjligt att endast kompilera om den modul som modifierats och på så sätt även spara in på kompileringstiden, utöver de besparingar som blir i överföringstid. En utvärdering av moduluppdelning på det system som använts i detta arbete är alltså intressant att genomföra.

8 Referenser

Binär Elektronik AB (publ). (1999). *Binär Elektronik – BiFas UHS*. [Online]. Tillgänglig från: http://www.binar.se/Products/hs_Uhs.htm [Hämtad 1999-02-16].

Burns, A. & Wellings, A. (1996). *Real-Time Systems and Programming Languages.* Addison Wesley Longman Limited.

Campbell, J. (1990). *Kommunikation med V.24/RS-232*. Gotab, Pagina International AB.

Halsall, F. (1996). *Data Communications, Computer Networks and Open Systems*. Fourth Edition. Addison-Wesley Publishing Company Inc.

Hancock, B. (1996). *Advanced Ethernet/802.3 Management and Performance*. Second Edition. Digital Press.

Lynch, T. (1985). *Data Compression Techniques and Applications*. Van Nostrand Reinhold Company, Inc.

Matsumoto, T. (1999). *LEO – Link Everything Online*. [Online]. Tillgänglig från: http://www.leo.org/pub/comp/os/dos/arcutils/execomp/diet145f.idx.html [Hämtad 1999-04-04]

PKWare, Inc. (1999). *PKWare, Inc. The Data Compression Experts!*. [Online]. Tillgänglig från: http://www.pkware.com/ [Hämtad 1999-04-04].

Pressman, R. (1997). *Software Engineering: A Practitioner's Approach*. Fourth edition. McGraw-Hill Companies, Inc.

Pocket Soft, Inc. (1999). *PocketSoft Home Page*. [Online]. Tillgänglig från: http://www.pocketsoft.com/ [Hämtad 1999-04-04].

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. (1991). *Object-Oriented Modeling And Design*. Prentice-Hall, Inc.

Salomon, D. (1998). *Data Compression The Complete Reference*. Springer-Verlag New York, Inc.

Williams, R. (1991). *Adaptive Data Compression*. Kluwer Academic Publishers.

Bilaga 1 – BiFas UHS

BiFas UHS är ett kombinerat positionerings- och PLC-system. Information om BiFas UHS har hämtats från intervjuer med anställda på Binär och från Binärs hemsida på internet.

Positioneringssystemet används för att kunna ställa olika servoaxlar i rätt läge. Med rätt läge menas då det läge som behövs för att kunna utföra den del i industriprocessen som är nödvändig för att kunna fortsätta arbetet. Detta kan till exempel vara att vrida en kniv 87,4 grader medurs.

PLC är en förkortning av Programmable Logic Controller. Ett PLC-system används till att styra olika automatiserade processer inom industrin.

BiFas UHS kan hantera ett flertal olika protokoll och in/utdata enheter. Systemet är uppbyggt kring ett halvlängds industri CPU-kort. Detta kort är i sin tur bestyckat med en AMD486DX4-100 CPU med 128 Kbyte cache. Primärminnet är på 8 Mbyte och till sekundärminne används en 8 Mbyte flash-disk. Operativsystemet som körs på kortet är MS-DOS 5.0. Utöver detta så används en 32-bitars DOS-extender för att få tillgång till allokering av större datastrukturer än 64 Kbyte. Anslutningsmöjligheter som finns att tillgå är en standard serieanslutning (RS-232) men en extra modul finns även att köpa till för att få tillgång till ethernet i form av en TP -anslutning.