Handelshögskolan vid Göteborgs universitet
Institutionen för informatik

Kommunikationsgränssnitt och dess tillämpning inom virtuella företagsväxlar.

Examensarbete 10 poäng ADB-programmet VT-1998
Författare: Fredrik Magnusson
Handledare: Wera Tegner Johansson

Abstrakt

Syftet med denna uppsats har varit att föreslå hur ett kommunikationsgränssnitt till en specifik produkt, Business Communications Manager, skall utformas. För att kunna göra detta har olika kommunikationslösningar, även kallat middleware, studerats med avseende på deras funktionalitet. Några av de tekniker som studerats är objektmäklare, transaktionshanterare och Remote Procedure Call (RPC). Dessutom har uppsatsen även beskrivit applikations-protokollen Hyper Text Transfer Protocol (HTTP) och Sockets. Ett av dessa protokoll valdes som kommunikationslösning, nämligen HTTP. Därefter utformades gränssnittet till systemet på grundval av ett antal centrala objekt som stödjer enkla operationer. Gränssnittets utformning, hur det skall anropas och vilka svar som fås är ingående beskrivet i uppsatsen och dess bilagor.

Utredningen kom fram till att det är möjligt att basera kommunikationen på HTTP, men att detta för med sig vissa specifika problem i samband med felhanteringen i systemet

Innehållsförteckning

1. Introduktion

1.1. *Inledning*

Ericsson Telecom i Mölndal utvecklar stödprodukter för administration, konfiguration och statistikbearbetning av virtuella företagsväxlar, även kallat CENTREX. Dessa är ett alternativ till de traditionella företagsväxlarna och innebär att företaget inte behöver investera i en egen företagsväxel utan i stället kan abonnera på en företagsväxeltjänst som administreras av en teleoperatör (t.ex. Telia). Enheten A/SG har utvecklat produktfamiljen BCM (Business Communication Manager) för hantering av CENTREX. BCM-produkten är utvecklad i Java i UNIX-miljö och har ett grafiskt användargränssnitt för att kommunicera med systemet. Nästa steg i utvecklingen av BCM är att möjliggöra kommunikation med systemet på andra sätt än genom det grafiska användargränssnittet. Anledningen till detta är att man i framtiden vill öppna upp för användarna av systemet (operatörerna) att kunna skapa egna klienter för olika ändamål i stället för att bara behöva arbeta med den stora klient som Ericsson tillhandahåller. För att åstadkomma detta behöver man skapa ett kommunikationsgränssnitt, dvs en beskrivning av hur systemet skall anropas, vad resultaten blir och vilka svar som kan förväntas. Det handlar med andra ord om att definiera ett protokoll för hur informationsutbytet med systemet skall ske. Det måste också finnas ett protokoll som kommunikation baserar sig på, såsom exempelvis HTTP, vilket är ett av de vanligast förekommande protokollen idag. Uppsatsen skall därför beskriva de vanligaste protokollen som man kan använda för att bygga distribuerade system.

Ett begrepp som i detta sammanhanget börjar bli allt vanligare är distribuerade komponentbaserade system, där komponenten är ett objekt som kan anropas från ett eller flera olika system. I företag idag finns en mängd olika datormiljöer och de objektsystem man skapar för en viss miljö kan inte återanvändas utan att behöva skrivas om. Problemet har varit avsaknaden av standarder. Under senare tid har dock försök gjorts för att få standarder inom detta område. Det talas speciellt om två sådana, DCOM och Corba och båda används idag av företag för att skapa distribuerade komponentsystem. Båda två har likheter, men skiljer sig också åt på avgörande sätt. Dessa kommer också studeras då de skulle kunna vara tillämpliga på BCM-produkten.

1.2.*Syfte och frågeställning*

BCM-produkten är idag ett väl fungerande standardsystem, som bygger på klient-server modellen. Nackdelen med systemet är dock att mycket av logiken ligger på klienten och kommunikationen mellan klienten och servern sker via ett egenutvecklat protokoll och det är därför i dagsläget omöjligt att utveckla nya klientsystem på ett snabbt och smidigt sätt. En önskvärd utveckling av produkten skulle därför vara att försöka flytta över affärslogiken till servern och förenkla sättet att kommunicera med systemet genom att skapa ett protokoll på en högre abstraktionsnivå.
För att kunna lyckas med detta behövs en modell för hur kommunikationen till systemet skall se ut och det behövs även ett tillförlitligt kommunikationsprotokoll att basera kommunikationen på. För att kunna utforma ett kommunikationsgränsnitt till BCM skall jag i denna uppsats beskriva de olika tekniker som finns för att bygga distribuerade system och jag skall sedan välja en av dessa tekniker och använda den på ett konkret tillämpningsfall.

Mitt problemområde utmynnar i dessa frågeställningar

- Villka tekniker finns tillgängliga idag för att bygga distribuerade system?
- Vilka tekniker kan tillämpas på BCM-produkten?
- Hur skall kommunikationsgränssnittet till BCM utformas, med avseende på underliggande protokoll och vilka operationer som skall vara tillämpliga?

Syftet med uppsatsen är att utreda hur ett kommunikationsgränssnitt skall utformas för att det skall bli så användbart som möjligt. Uppsatsen skall också försöka beskriva de olika tekniker som finns för att implementera ett kommunikationsgränssnitt. Detta skall ske dels genom litteraturstudier och dels genom att studera ett konkret användningsfall. Den tänkta målgruppen är personer som utvecklar klient-server system och speciellt de som arbetar med BCM-produkten.

1.3.*Avgränsningar*

Implementationerna av modellerna kommer bara beröra en delmängd av systemet, nämligen operationer som gäller de centrala objekten "business group","extension" och "number serie" .
Jag kommer inte att behandla något om de transportprotokoll som ligger under det applikationsprotokoll jag kommer att utforma, utan jag nöjer mig med att konstatera att detta utgörs av TCP/IP stacken.

1.4.*Metod*

Anledningen till att jag valde att skriva om detta ämne är att jag tidigare har gjort praktik vid Ericsson och fick då jobba med applikationsutveckling inom ramen för BCM. Jag fick

under den tiden ganska stor inblick i systemet och det var därför jag fick möjlighet att göra detta examensarbete. Svårigheten med denna uppgift har varit att kunna kombinera skolans krav på en vetenskaplig uppsats och en mycket praktiskt orienterad uppgift. Mitt tillvägagångssätt har därför varit starkt präglat av den praktiska tillämpning jag har gjort. För att kunna svara på mina frågeställningar har jag valt att dels använda mig av litteraturstudier och dels få svar genom att observera resultatet av min praktiska tillämpning. Jag har under arbetets gång upptäckt att det har varit svårt att hitta någon specifik litteratur som behandlar ämnet. Jag har därför förgäves försökt hitta nya infallsvinklar på problemområdet. Orsaken till avsaknaden av litteratur är troligtvis att problemområdet starkt hänger samman med det specifika system som man skall införa kommunikationsgränsnittet till.

2. Kommunikationsgränssnitt

2.1.*Definition*

Ett gränssnitt definieras i svenska ordlistan som en förbindelse mellan olika enheter i ett datasystem. Ett kommunikationsgränsnitt är med andra ord en uppsättning regler för hur andra system skall anropa det aktuella systemet. Dels så är det regler för hur själva överföringen av anropet sker och dels är det vilka parametrar som skall skickas och hur det anropade systemet kan svara. I takt med att datorsystem blir allt mer distribuerade och utspridda över organisationer ökar behovet av att dessa skall kunna samverka på ett felfritt och framförallt transparent sätt gentemot användaren. Detta i sin tur ställer höga krav på kommunikationsgränssnittet.

2.2.*Gränssnitt*

Själva gränssnittet är systemets ansikte utåt och är det som definierar vad som kan göras i systemet. Det kan också gå under beteckningen API (Application Programming Interface) och som namnet antyder är det till för att underlätta utvecklingen av applikationer för olika tillämpningar. Oftast när man talar om APIer avses en utvecklingsmiljö som ligger på samma dator som klienten.
Ett annat mycket välkänt gränssnitt är SQL som är en standard för kommunikation med relationsdatabaser. Anledningen till att SQL har blivit så populärt är att det är ett procedurspråk, dvs det säger vad som skall göras men inte hur, det är upp till varje databastillverkare att själv implementera. Det går också att snegla på området objektorienterad design där ett objekts publika gränsnitt huvudsakligen är till för att man i efterhand skall kunna gå in och ändra på objektets interna struktur utan att påverka andra objekt som är beroende av det.
Varför vill man ha ett kommunikaionsgränssnitt till ett system? Dels för att systemet skall kunna samverka med och skicka data till andra tillämpningar, men även också för att lätt kunna utveckla nya tillämpningar som använder sig av det befintliga systemet.
Vilka krav skall då ställas på ett gränssnitt? Först och främst skall det vara väldokumenterat hur anrop skall ske och vad som returneras. Det skall också tydligt anges vilka delar av systemet som påverkas av ett anrop. Vidare skall det vara lätt att förstå, konsekvent och stödja grundläggande operationer. Syftet med ett gränssnitt är att dölja den inre strukturen genom att stödja operationer på hög abstraktionsnivå.
För att ett gränssnitt skall vara användbart måste det kunna anropas från ett annat system. Därav namnet kommunikationsgränsnitt.

Ett kommunikationsgränsnitt skall därför bestå av två delar:

1. Tydliga operationer, vilket innebär att anrop och svar är väl dokumenterade. Hur gränssnittet skall utformas, dvs vilka operationer det skall stödja är beroende av det underliggande systemets struktur. Dock kan man rent generellt säga att det är fördelaktigt om gränssnittet stödjer enkla operationer.
2. Ett protokoll att basera kommunikationen på. Här finns flera olika alternativ att välja på och dessa kommer jag studera närmare i min rapport.

3. Klient-Server

(Nielsen 1995)
Ett vanligt sätt att bygga system idag är enligt den sk klient-server modellen. Denna modell innebär att ett systems centrala delar, t.ex. databasen, ligger på en server som tar emot anrop från en klient. Servern utför de bearbetningar som behövs och skickar sedan tillbaka ett svar. Detta bygger på att det finns ett standardiserat sätt att kommunicera mellan klient och server. Den bakomliggande tanken är alltså att en applikation delas upp i sina delkomponenter, som sedan kan hanteras av olika datortyper. Detta innebär en stor skillnad mot hur administration av applikationer och datahantering tidigare utfördes på centrala system (primärt stor- och minidatorer) förr i världen. I ett centralt system fanns således samtliga delar av applikationen i en och samma dator.
Anledningen till att jag har valt att studera klient-server modellen är att den idag är det i särklass vanligaste sättet att bygga system på. Det är dessutom så att man kan betrakta BCM-produkten som ett klient-server system där servern motsvaras av AXE-växeln, vilken har ett gränssnitt i form av MML-kommandon.
De främsta fördelarna med klient-server modellen kan sammanfattas på följande sätt.

Snabbare applikationsutveckling Klient-server applikationer utvecklas ofta med databasorienterade fjärdegenerationsverktyg (4GL-verktyg). Dessa avancerade högnivåverktyg utgör en lättillgänglig utvecklingsmiljö där applikationerna ofta kan byggas på en fjärdedel av den tid som man tidigare behövde för att bygga applikationer på i en mer centraliserad miljö. Oftast är även de fasta kostnaderna som licensavtal och löner till programmerarna lägre i samband med fjärdegenerationsverktyg jämfört med traditionella programmeringsverktyg för centrala system.
Flexibilitet En av klient/server-teknikens grundpelare är att en applikation skall vara enkel att flytta från en plattform till en annan. Detta innebär stora ekonomiska fördelar såväl i utvecklingsfasen som administrationsfasen för de företag som använder flera olika plattformar.
Grafiska användargränsnitt
En grafisk användarmiljö innebär många inlärnings- och produktionsmässiga fördelar. Med hjälp av 4GL-verktyg kan man snabbt utveckla prototyper som sedan kan provas på användaren, vilket gör att denne har stort inflytande över hur gränssnittet utformas.

Detta ger en bakgrund till modellen klient-server, som är viktig att ha med sig när vi studerar middleware.

4. Middleware

(International Systems Group 1997)
Detta är ett begrepp som håller på att bli allt vanligare. Med middleware avses här ett system som ligger mellan klienten och servern och som innebär att klientprogrammet kommunicerar med servern via ett mellanskikt och därför inte behöver bry sig om detaljer kring var servern befinner sig m.m.
Den traditionella tvåskikts klient/server-arkitekturen har en del begränsningar såsom problem med skalbarhet, säkerhet, "feta" klienter och portabilitet. De flesta av dessa problemen kommer av att klienten är för beroende av servern. Det främsta syftet med middleware är därför att koppla loss klienten från servern.
Det finns flera fördelar med den treskiktsmodell som middleware innebär. De olika applikationskomponenterna kan placeras där det är mest kostnadseffektivt att exekvera dem (t.ex. nära databasen), de kan exploatera användargränssnittet fullt ut och de kan integreras med existerande applikationer, etc. Detta möjliggör för applikationer att användas över hela företaget och inte bara vara begränsade till en viss avdelning.
Utveckling och distribution av dessa nya treskikts-applikationer ställer en rad nya krav på utvecklingsmiljön. De viktigaste är:

- Systemutvecklare behöver *flexibla*, men ändå *robusta* kommunikations-APIer för att möjliggöra för applikationer att med lätthet utbyta information på ett synkront och asynkront sätt.
- Det behövs en gemensam och *konsistent* miljö som gör det möjligt för applikationsutvecklarna att koncentrera sig på att lösa affärsproblem snarare än att behöva bry sig om underliggande hårdvaruplattform, operativsystem och kommunikationsprotokoll.
- Det är också viktigt att i en distribuerad applikation försöka ha *löst kopplade komponenter* som bara kommunicerar via gränssnitt. Detta medför att påverkan på systemet blir betydligt mindre om en komponent modifieras eller ersätts.
- Ett annat viktigt krav är *lokalitetstransparens* vilket innebär att en applikation inte behöver bry sig om var i den distribuerade miljön en annan applikation befinner sig. Detta bör ske med hjälp av någon katalogtjänst.
- Distribuerade applikationer måste vara *säkra*. Användare skall kunna lita på den identifikation och auktorisation av klienter och servrar som finns och även att meddelanden som skickas över nätverket är skyddade från avlyssning.
- Det är också viktigt att applikationer är *skalbara*, eftersom detta var ett av problemen med första generationens klient-server lösningar.
- Utvecklingsmiljön skall också kunna erbjuda tjänster såsom *feltolerans* och *återhämtning* efter t.ex. systemkrascher, eftersom i takt med att applikationer blir alltmer distribuerade kan nästan varje komponent vara en potentiell felkälla.

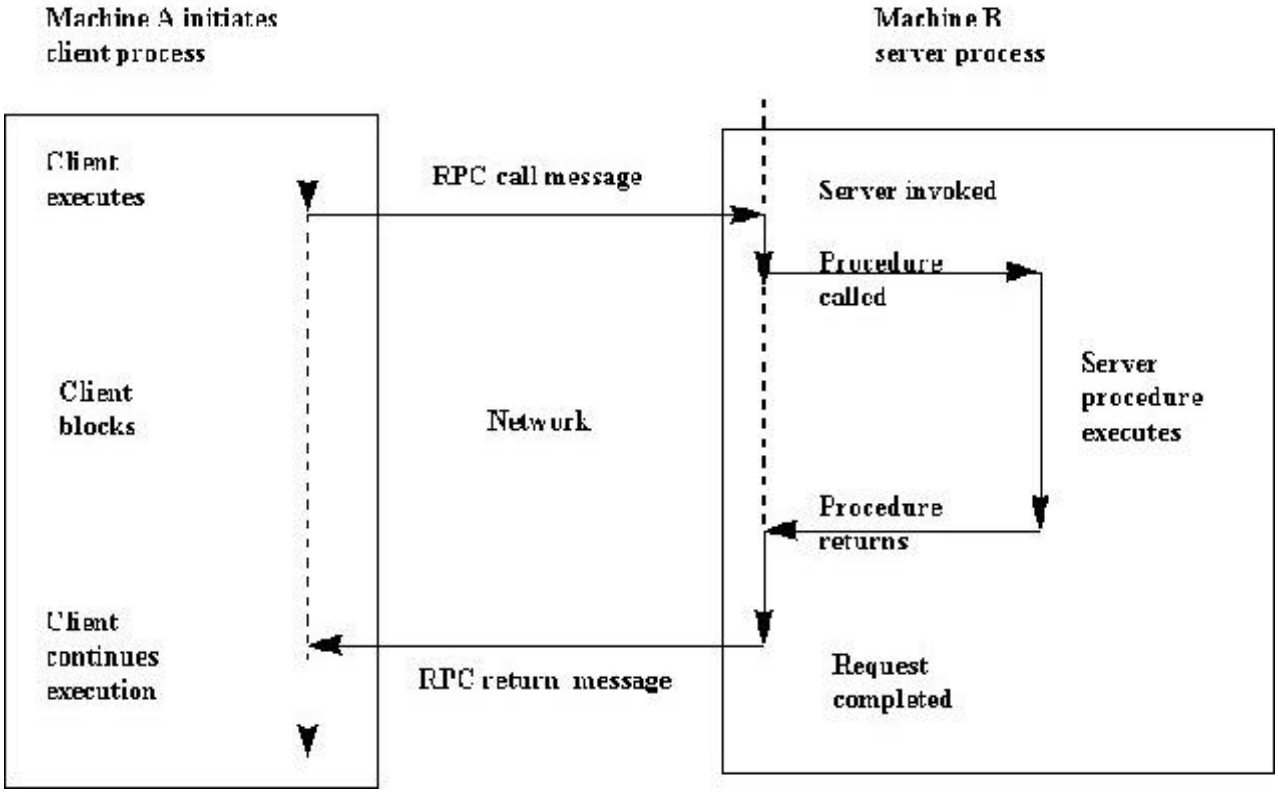
Det finns idag kommersiella produkter som uppfyller dessa krav. Dessa kan i sin tur delas in i olika kategorier av middleware:

- Middleware baserad på Remote Procedure Calls (RPC).
- Meddelandeorienterad middleware (MOM)
- System för transaktionshantering
- Objektorienterad middleware även kallat objektmäklare

Nedan följer en kort beskrivning av några sådana system.

4.1.*RPC*

(International Systems Group 1997)
Remote Procedure Calls (RPCs) är en av de mognaste mekanismerna för att bygga distribuerade applikationer. RPC är utvecklad i UNIX-miljö och är idag en vida accepterad teknik. Det finns idag ett par middleware-produkter som baserar sig på RPC, OSF's DCE och SUNs ONC. Enkelt förklarat kan man säga att RPC innebär att en procedur eller funktion i ett program kan anropas fast den ligger på en annan dator.



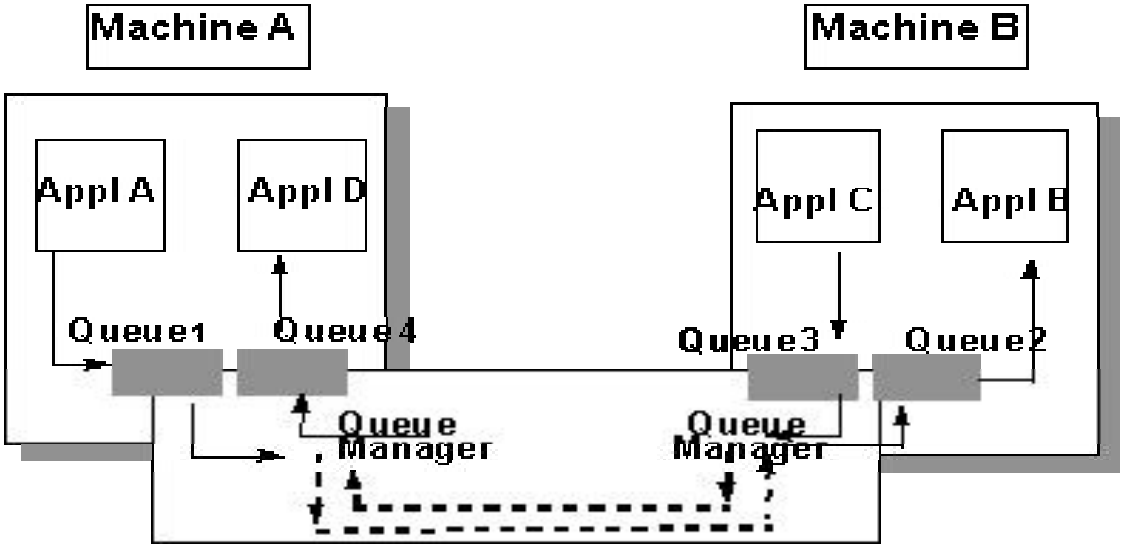
Beskrivning av RPC (International Systems Group 1997)

För att kunna koppla samman klienten med serverkomponenten via RPC som middleware är det nödvändigt att varje funktion som klienten kan anropa skall representeras av en

"stubbe", en slags platshållare för den riktiga funktionen på servern. Denna stubbe ser ut som fjärrfunktionen och ger lokalitetstransparens för klienten. Därför är det ganska enkelt att använda RPC-modellen för att bygga distribuerade applikationer. Klienter anropar en fjärrfunktion på samma sätt som en lokal funktion.
Dessutom behöver utvecklaren inte tänka på synkronisering mellan en serie anrop, eftersom varje anrop överför kontroll till server processen och klientprocessen är blockerad tills svaret från servern är överfört till anroparen (även kallat synkroniserad eller blockerande kommunikation).
På serversidan fungerar det på exakt samma sätt. Här anropar ett stubbprogram proceduren på servern på exakt samma sätt som det skulle ha anropats av klientprogrammet om det hade funnits inom klientprocessen. Applikationskomponenten på servern kräver dock en del merabete jämfört med klientdelen för att göra sina tjänster synliga för klienten på nätverket. Applikationen måste bl.a registrera sig hos katalog- och säkerhetstjänsten för att allt skall fungera.
Även om stubb-programmen ser ut som den riktiga proceduren på klienten, innehåller de inte någon affärslogik. Istället innehåller de logik för att ta emot parametrarna för ett funktionsanrop, paketera dem för transporten över nätverket och utföra motsvarande operation för svaret från servern.
I de flesta fall innebär RPC ett statiskt förhållande mellan de olika komponenterna i en distribuerad applikation. Detta innebär att när klientapplikationen väl har blivit kompilerad och länkad med RPC-stubbarna i en körbar modul, så är dess bindning till en viss serverprocess cementerad.
Översättningen av datarepresentationen som beror på hårdvaran och nätverket innebär en omvandlingsprocess. Denna process behöver en komplett beskrivning av alla data som är involverade i ett anrop inklusive dess typ, format och längd. Denna beskrivning måste tillhandahållas av utvecklaren av serverapplikationen i form av IDL (Interface Definition Language). IDL är ett universellt högnivåspråk som kan definiera gränssnitten vilka representerar kontrakten mellan klient- och serverapplikationer. Gränssnittet består av funktionsnamn, parametrarna som skickas mellan klienten och servern och möjligtvis en resultattyp. IDL är oberoende av programmeringsspråket vilket innebär att klienten inte behöver bry sig om det specifika anropssättet för ett visst programspråk. Så länge det finns en omvandlingsfunktion för IDL till det aktuella programmeringsspråket så kan man använda vilket programspråk som helst. När IDL väl är klart används det som indata för en kompilator som genererar stubbrutinerna som skall anropas av klientprogrammet och som i sin tur anropar serverstubben.

4.2.*Meddelandeorienterad middleware*

(International Systems Group 1997)
Meddelandeorienterad middleware (MOM) innebär en process där data och kontroll distribueras genom ett utbyte av meddelanden. Meddelanden är strängar av data som betyder något för applikationerna som utbyter dem. Förutom applikationsrelaterad data, kan meddelanden inkludera kontrolldata som endast är relevant till meddelande-hanteringssystemet. Den informationen kan användas för att lagra, vidarebefordra, leverera och spåra data som tidigare har skickats. Ett annat viktigt drag är att meddelandeorienterad middleware stöder både synkron och asynkron kommunikation och är mer en händelsestyrd snarare än en procedurell process.
Utbyte av meddelande mellan program är en direkt kommunikationsmodell. Genom att skicka ett meddelande görs en applikationsförfrågan från ett program till ett annat. Båda programmen kommunicerar med varandra direkt på ett förbindelseorienterat sätt. En logisk förbindelse mellan programmen behöver upprätthållas för att det hela skall fungera.



Beskrivning av ett köhanteringssystem (International Systems Group 1997)

Meddelandeutbyte innebär generellt att kommunikationsmekanismen antingen kan vara synkron (dvs sändaren är blockerad tills mottagaren skickar tillbaka ett meddelande - som RPC) eller asynkront (genom att kontrollera med jämna mellanrum eller via anrop tillbaka) , vilket gör denna middleware lämplig för händelsestyrda applikationer.
Meddelandeköer är en indirekt program-till-program process som möjliggör för program att kommunicera via meddelandeköer snarare än att anropa varandra direkt.
Meddelandeköer innebär alltid förbindelselös modell och därför är det inte nödvändigt att det andra programmet är närvarande. Detta möjliggör för program att fortlöpa oberoende, med olika hastigheter och utan en logisk koppling mellan dem. Meddelanden placeras i en kö (som antingen är minnes- eller diskbaserad) för antingen omedelbar eller senare

leverans. För att hantera dessa meddelandeköer behövs någon form av lokal köhanterare, som hanterar lokala köer och garanterar att meddelandena kommer fram. Den kan också ha andra tjänster som prioritering av vissa meddelanden och lastbalansering.
Meddelandeköer brukar också innebära stöd för en rad olika kvalitetstjänster:

- Tillförlitlig meddelandeleverans, under utbytet av meddelande förloras inga nätverkspaket.
- Garanterad meddelandeleverans, meddelanden levereras till destinationsnoden antingen direkt utan fördröjning, eller eventuellt med fördröjning om nätverket eller destinationsnoden inte är tillgängligt för tillfället. I det senare fallet garanterar systemet att meddelanden levereras om nätverket/noden är tillgängligt inom en viss tidsperiod.
- Ingen duplicering av meddelanden.
- Meddelandeköer kan antingen vara feltolerant eller icke feltoleranta. En feltolerant meddelandekö kan återställas om det blir något fel på köhanteraren, vilket innebär att kön hela tiden måste sparas på disk för att detta skall vara möjligt.
- En del produkter för meddelandeköer stödjer också så kallade utlösare. Då ett speciellt meddelande anländer till en kö startas ett applikationsprogram om det inte redan är igång. Detta kan minska förbrukningen av resurser då en applikation kan vara igång endast då det finns arbete att utföra.

4.3. *Transaktionshanterare*

(International Systems Group 1997)
En transaktionshanterare är en slags middleware som används för att övervaka transaktioner. En transaktion är mer än bara en enkel uppdatering av en post eller fil. Det är snarare en serie steg som är nödvändiga för att kunna bearbeta en interaktion med en kund. Den främsta anledningen för att använda en transaktionshanterare är om något går fel i en del av en transaktion så återkallas hela transaktionen av transaktionshanteraren. De transaktionshanterare som har utvecklats använder sig enbart av synkroniserad transaktionshantering för databaser.
En riktig transaktion måste uppfylla fyra egenskaper för att kunna hanteras. Dessa är:

- *Atomär*, alla operationer som en applikation utför som inkluderar uppdateringar av olika resurser grupperas i en så kallad arbetsenhet. Denna enhet anses vara atomär, dvs den går inte att dela på. Antingen utförs alla operationer i enheten eller så utförs ingen. Om en transaktion måste avbrytas på grund av systemfel eller liknande, skall de operationer som har hunnit utföras återkallas.
- *Konsistens*, i slutet av en transaktion måste alla resurser som har deltagit i transaktionen vara i ett konsistent tillstånd
- *Isolering*, samtidig åtkomst till resurser av olika arbetsenheter koordineras så att de inte påverkar varandra. Med andra ord, arbetsenheter som konkurrerar om resurser isoleras från varandra.
- *Hållbarhet*, alla operationer som har utförts under en transaktion skall vara fullt genomförda och lagrade.

I de flesta databassystem finns idag funktioner för transaktionshantering, så det är möjligt att använda sig av någon annan kategori av middleware och låta databasen sköta transaktionshanteringen. Men om applikationen använder sig av flera databaser måste i så fall transaktionslogiken ligga på klienten och då kan det vara bättre att använda en transaktionshanterare.

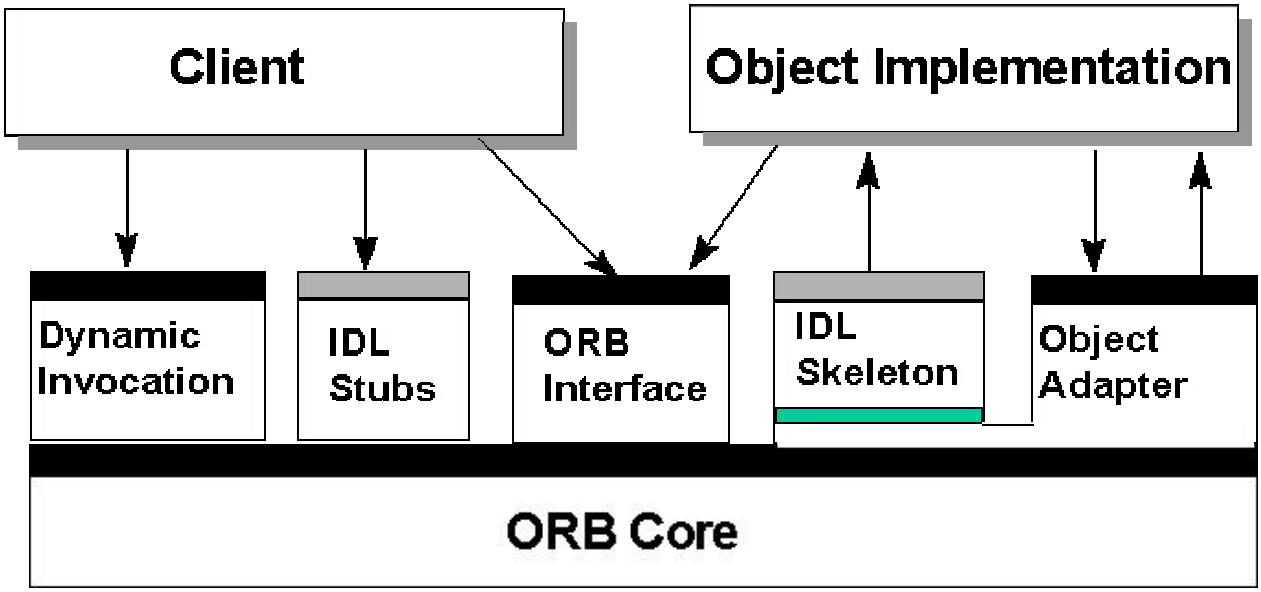
4.4.*Objektorienterad middleware*

(International Systems Group 1997)
Fram tills nu har jag bara beskrivit olika sorters funktionella middleware ej baserat på objektorientering. Det blir dock allt vanligare att organisationer använder sig av objektorientering för sin systemutveckling. För att kunna distribuera sina applikationer behöver de en middleware som håller reda på alla objekt trots att de är utspridda. Objektiorienterad middleware, även kallat ORB (Object Request Brokers) är uppdelat i två kategorierer. Produkter som baserar sig på CORBA (Common Object Request Broker Architecture) standarden från OMG (Object Management Group) och Microsofts' DCOM

4.4.1. Corba

Corba är en samling specifikationer för hur objekt skall samverka. Dessa specifikationer definierar hur objekt definieras, avlägsnas, anropas och hur de kommunicerar med varandra. Ett centralt begrepp i Corba är något som kallas objektmäklare (ORB).
En objektmäklare kan ses som en typ av middleware, som möjliggör för ett klientobjekt att anropa ett serviceobjekt på en server. Det här är likartat för vad som har beskrivits för RPC-teknologin. Och precis som i fallet med RPC är en nyckelkomponent IDL, det specifika språk som dokumenterar gränssnittet mellan klienter och servrar. I CORBA-fallet har dock IDL-språket utvecklats för att stödja koncepten i objektorientering (t.ex arv).

En ORB har följande utseende.



Beskrivning av CORBA-arkitekturen. (International Systems Group 1997)

Bilden ovan visar de olika sätt som klienten kan anropa objekt på. På samma sätt som i RPC använder man sig av sk stubbar på klient- och serversidan, vilket innebär ett statiskt förhållande mellan objekten. Kommunikationen mellan klient och server är synkron, dvs klienten är blockerad tills den får ett svar. Det finns dock möjlighet att även anropa objekt dynamiskt under körning vilket är en väldigt kraftfull möjlighet. Detta är dock något som innebär att det måste finnas ett register där alla objekten finns registrerade, så att klienten kan söka efter ett visst objekt under körning.
Specifikationerna för Corba specificerar också vissa tjänster som en ORB-implementation kan erbjuda. Det kan vara tjänster som rör säkerhet, händelshantering, transaktionshantering, synkronisering m.m.
Corba specificerar ingenting om hur en ORB skall implementeras, utan säger bara vilka gränssnitt den skall ha. Det finns därför lika många implementationer som det finns Corba-produkter på marknaden.

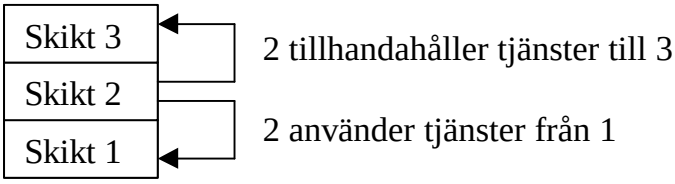
4.4.2. DCOM

Till skillnad från Corba som kan anses vara en öppen standard finns det en konkurrerande standard för distribuerade objekt som ägs av Microsoft och kallas DCOM. Denna produkt utvecklas som sagt av Microsoft och de har bara utvecklat den för Windows-plattformarna (NT, Win95) och låter andra mjukvaruföretag utveckla för andra plattformar som Unix, Open MVS och VMS m.fl (Datateknik 98-02 Objekt på näten).
DCOM som står för Distributed COM är en produkt som bygger på Microsofts COM-teknik som har funnits i flera år. COM står för Component Object Model och är en modell

som specificerar hur objekt skall definieras, avlägsnas, anropas och hur de kommunicerar med varandra. Den har också en egen implementation av IDL-språket.
DCOM är en uppgradering av COM, med en specifikation som omfattar säkerhetshantering och olika åtkomstmetoder för nätverket. DCOM är helt enkelt COM med ett nätverksprotokoll. För att säkerställa kommunikationen används RPC-anrop mellan klient och server. Detta innebär förstås att klienten är blockerad tills ett svar kommer.
Nackdelarna med DCOM jämfört med Corba är att den ägs av ett företag. Dock används den av många oberoende mjukvaruföretag och den spelar även en stor roll för många andra teknologier från Microsoft, så den har blivit en de facto standard. Då båda Corba och DCOM är väldigt viktiga tekniker som kan spela stor roll i framtiden har det dykt upp program som gör att ett COM-objekt kan kommunicera med ett Corba-objekt och vice versa. Troligtvis kommer dessa standarder att smälta samman i framtiden.

5. Kommunikationsprotokoll

Jag har nu beskrivit olika tekniker att basera kommunikationen på hög nivå i form av middleware. Gemensamt för alla dessa tekniker är att de baserar sig på någon form av kommunikationsprotokoll. Det kan därför vara intressant att kort beskriva hur ett sådant fungerar.
Kommunikation mellan datorer är en komplicerad process som kräver en stabil kommunikationsmiljö för att fungera. Dels krävs ett nät för att kunna överföra data och framförallt krävs protokoll som bl.a sköter felhantering, gör omvandlingar mellan olika dataformat och som styr flöden i nätverket. Denna process är för det mesta osynlig för användaren, men märks förstås om det t.ex blir ett avbrott på nätet någonstans.
Utvecklingen inom datakommunikationsområdet har idag kommit väldigt långt och hastigheten och tillförlitligheten i nätverken stiger alltmer. En av de främsta anledningarna till detta är att det har vuxit fram en rad standarder inom datakommunikationsområdet.
Att kommunicera mellan två enheter är en ganska komplex företeelse och därför blir även protokollet som styr kommunikationen väldigt komplext. (Andréasson, Carlsson 1997, s. 26-28) För att minska komplexiteten av programvaran som styr kommunikationen, indelas kommunikationen i ett antal nivåer eller skikt. Tillsammans bildar skikten en logisk enhet som sköter kommunikationen och varje skikt har ett eget protokoll som ger service till närmaste övre skikt och utnyttjar tjänster från närmaste underliggande skikt. Längst ned bland skikten finns den fysiska förbindelsen och högst upp applikationen.
Kommunikationen mellan skikten sker enligt fasta regler som kallas gränssnitt. Grässnittet är den enda kunskap som finns om ett angränsande skikt. Därför sägs skikten vara transparenta.



Det här sättet att bygga skikt på varandra brukar benämnas protokollstack. (Elbert, Martyna 1994, s 40-43) Varje lager i stacken kommunicerar med motsvarande lager i en annan stack. Som tidigare nämnts sker kommunikation via meddelanden. För att meddelandet skall kunna skickas ut på nätverket måste det passera genom varje lager. Varje lager lägger då på kontrollinformation för respektive protokoll och på motsvarande sätt plockas denna kontrollinformation bort då meddelandet går igenom den mottagande stacken.

5.1.*TCP/IP*

(Elbert, Martyna 1994, s 50-53)
Denna protokollstack har idag en stor utbredning pga av Internets utveckling och att den är fritt tillgänglig i Public Domain. Detta har gjort att TCP/IP numera kan betecknas som en öppen standard. Det gränsnitt som jag senare skall tillämpa baserar sin kommunikation på TCP/IP. Fördelarna med TCP/IP som nätverksprotokoll kan sammanfattas:

- Öppenhet. Specifikationerna är gratis tillgängliga
- Tillgänglighet. Implementeringar av TCP/IP är tillgängliga till låg kostnad och ingår som en del i de flesta operativsystem
- Enkelhet. Under dess utveckling har hela tiden prototyper utvecklats när problem har uppkommit. Dessa har sedan fått genomgå en ganska informell standardiserings-process. Fördelen är att fungerande lösningar ständigt är tillgängliga för distribuerade applikationer och att de är väl beprövade av internetanvändare.

5.2.*Sockets*

(Elbert, Martyna 1994, s 59)
En viktig del i utvecklingen av TCP/IP är en hög tillgänglighet av APIer för att underlätta applikationsutvecklingen. Ett bra exempel på en sådan är Sockets som är baserat på en abstraktion av en ändpunkt på en tvåvägskommunikation och används för överföring av ström-orienterad data via t.ex TCP. Med ström-orienterat menas att datan inte har någon fast postlängd. Sockets är en uppfinning av Berkley UNIX och tillämpas idag i flera icke-UNIX operativsystem också. Ett Socket-API stödjer ett antal enkla operationer, såsom *read,write,connect* etc.

5.3.*HTTP*

(James Marshall 1997) HTTP skapades av grundaren till Internet, Tim Berners-Lee, som insåg att det behövdes ett protokoll för att hantera alla filtyper på det globala nätverket. Det har använts på Internet sedan 1990 och är under ständig utveckling

HTTP står för HyperText Transfer Protocol. Det är nätverksprotokollet som används för att hämta filer och annan data (även kallat resurser) på World Wide Web, antingen det är HTML-filer, bildfiler, frågeresultat eller någonting annat. För sin kommunikation använder sig HTTP normalt av TCP/IP sockets.
HTTP använder sig av klient/server-modellen för sin kommunikation, dvs klienten skickar en förfrågning till en server som ger ifrån sig ett svar. HTTP-servern lyssnar normalt på port 80.
Det är ett icke förbindelseorienterat protokoll, dvs att efter server har skickat ett svar på klientens förfrågan kopplas förbindelsen ned och servern "glömmer" allt om klienten. Detta skall jämföras med exempelvis FTP-protokollet som hela tiden upprätthåller en förbindelse mellan klienten och servern eftersom klienten t.ex skall kunna navigera i serverns filsystem.
Som följd av det tidigare nämnda kan man också kategorisera HTTP som ett tillståndslöst protokoll, dvs det finns inget minne mellan klientförfrågningarna, så en ren HTTP server behandlar varje förfrågan som om det vore en ny klient utan något sammanhang. Det finns dock en viss möjlighet att med hjälp av så kallade ”cookies” behålla ett tillstånd mellan sessionerna.

Protokollet använder sig av textbaserad kommunikation baserat på US-ASCII
Hur ser då själva kommunikationen ut? Klienten och servern kommunicerar via meddelanden. Meddelandet är uppbyggt ungefär på samma sätt oavsett om det är klient eller server som skickar det. Ett meddelande har följande struktur:

<inledande rad, olika för klient respektive server>
Header1: värde1
Header2: värde2
Header3: värde3
<tom rad>
<en valfri meddelandekropp som t.ex. kan innehålla en fil eller data från en fråga>

Man kan maximalt ha 16 olika headers i HTTP1.0 men ingen är nödvändig. I HTTP1.1 kan det finnas 46 olika headers och en är nödvändig (host). Den inledande linjen och headern skall avslutas med en CRLF (ASCII värdena 10 och 13)
Klienten kan begära resurser av servern med olika metoder. Den vanligaste metoden är GET som i princip säger "ge mig denna resurs". Ett par andra metoder är POST och HEAD.

En klientförfrågan består av tre delar separerade av mellanslag och har följande utseende:
< metod sökväg version av HTTP >
Ex.
GET /path/to/file/index.html HTTP/1.0

Svaret från servern kan se ut på följande vis
< version statuskod meddelande >
Ex.
HTTP/1.0 200 OK

Statuskoden består av tre siffror och den första siffran indikerar vad som är fel.
1xx indikerar ett informations meddelande
2xx indikerar framgångsrik bearbetning
3xx omdirigerar klienten till en annan adress
4xx indikerar ett fel hos klient
5xx indikerar ett fel hos servern

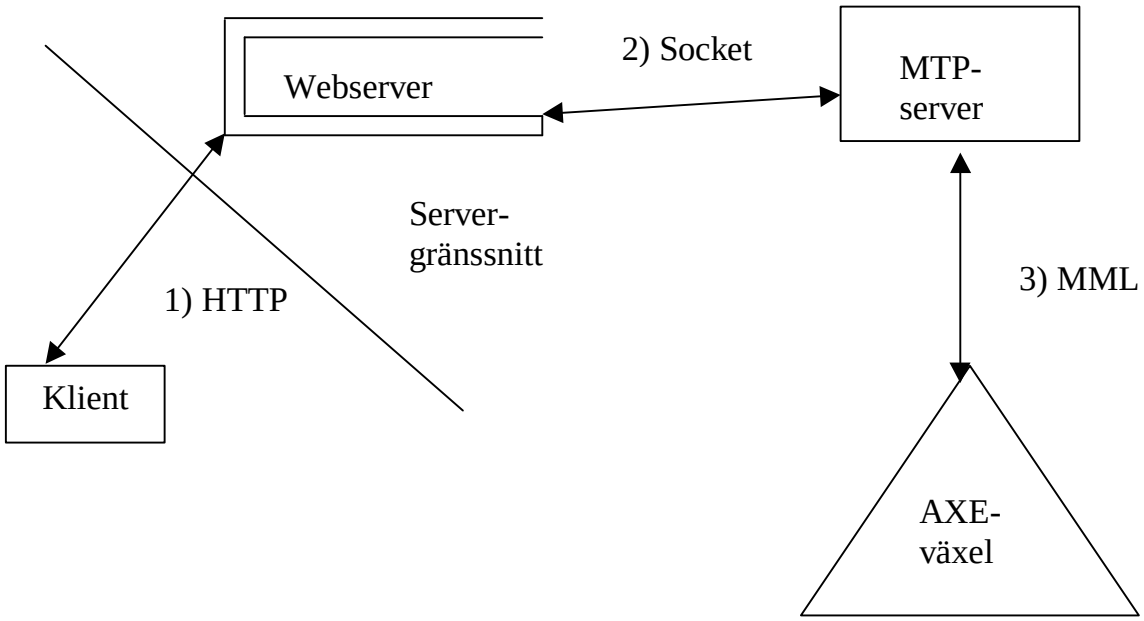
För att kunna skicka med information om ett anrop eller svar använder man sig av sk header-rader. Dessa ser ut på följande sätt: "Header-namn: värde" CRLF. Det finns ett antal fördefinierade ”headers”, som alltid bör finnas med. Efter alla ”headers” kan det följa med en meddelandekropp, som helt enkel är data av något slag. För att beskriva denna data används en header som har namnet "Content-Type"
Som jag nämnde tidigare finns det två andra metoder förutom GET för att skicka HTTP-anrop. Den ena är HEAD och fungerar precis som GET förutom att den bara returnerar header-linjer. Den andra är POST som används för att skicka data till HTTP-servern för vidare bearbetning. Det uteslutande vanligaste användningsområdet är för att skicka data till ett CGI-program. CGI står för Common Gateway Interface och innebär att man via en webserver kan köra ett vanligt program. Ett annat sätt att skicka data i samband med ett anrop är att helt enkelt lägga till det efter sökvägen åtskilt med ett frågetecken.
Ex. GET path/to/file/BCM_EI?OBJECT=BG HTTP/1.0

En viktig aspekt för att välja HTTP som kommunikationsprotokoll är att det numera finns med i flera programspråk, t.ex. Java.

6. Kommunikationsgränssnitt till BCM

6.1.*Grundläggande begrepp*

BCM-systemet är som tidigare nämnts ett system för att hantera virtuella företagsväxlar. Den virtuella växeln är fysiskt implementerad i en AXE-växeln och för att kunna kommunicera med en AXE-växel har Ericsson konstruerat ett gränssnitt som kallas Man Machine Language. Det är kring detta gränssnitt systemet är uppbyggt. Det är ett klient/server system utvecklat i Java där klienten kan köras antingen som en applet eller som en fristående applikation med hjälp av Java Virtual Machine. I själva verket är det ett klient-server system, där servern består av flera delar. I dagsläget är system uppbyggt på följande speciella sätt.



1. Klienten översätter det användaren gör till MML-kommandon som sedan skickas via HTTP till ett CGI-program (en Java servlet) på webservern för vidare bearbetning.
2. Detta program gör vissa bearbetningar och kopplar sedan upp sig via en socket till en speciell server som kommunicerar direkt med AXE-växeln.
3. Denna process känner till all logik som gäller för att skicka MML-kommandon och kommunicerar direkt med AXE-växeln via ett kommunikationsprotokoll som heter X.25.

Svaret från AXE-växeln går sedan tillbaka samma väg ända till klienten. Som systemet är utformat nu måste alltså klienten känna till logiken med MML-kommandon och det är därför som det är svårt att på ett enkelt sätt utforma nya klienter. Klientdelen är dock uppbyggd på så sätt att man har separerat grafiska komponenter och de komponenter som bygger upp och skickar kommandon till webservern. Detta gör det möjligt att tämligen enkelt separera dessa och ändå bibehålla all funktionalitet i systemet, eftersom de innehåller all logik. En nödvändig utveckling av BCM är nu att det skall vara möjligt att kommunicera med systemet på ett lättare sätt än via MML-kommandon.

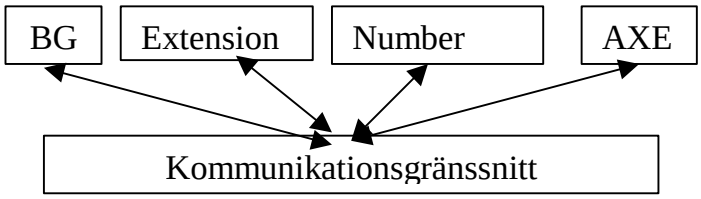
6.2.*Översikt*

Målet med min uppgift har varit att möjliggöra kommunikation med BCM-produkten på annat sätt än genom det grafiska gränssnitt som används idag. Det är viktigt att detta fungerar på ett pålitligt sätt och att en applikationsprogrammerare snabbt skall kunna sätta sig in i systemet. Efter att ha studerat de olika tekniker som finns för att implementera kommunikationsgränsnitt har jag valt att basera min lösning på HTTP som kommunikationsprotokoll, eftersom detta möjliggör en utvecking av antingen webbaserade eller vanliga klienter. Eftersom systemet inte kräver att något tillstånd sparas mellan sessionerna lämpar sig HTTP-protokollet bra eftersom det är tillståndslöst. Då systemet sedan tidigare använder sig av en webserver innebär det också att en befintlig resurs kan användas för att möjliggöra kommunikationen. Själva gränssnittet är utformat på så sätt att jag har försökt identifiera ett antal centrala objekt som operationer skall kunna utföras på. Operationerna är ganska få till sitt antal och har enkla namn som "CREATE, UPDATE" m.m.
Min modell kommer fungera på följande sätt:

1) Systemet nås genom en webserver.
2) Klienten skickar ett HTTP-anrop till mitt program på webservern. I URL-strängen anges en rad parametrar som är nödvändiga för att anropet skall kunna bearbetas
3) Programmet utför instruktionerna och returnerar ett svar via HTTP enligt ett fördefinierat protokoll.

6.3.*Systembeskrivning*

BCM är som jag tidigare nämnt helt objektorienterat. För att läsaren skall få lite större inblick i systemet tänkte jag här kort beskriva de viktigaste klasserna. En virtuell företagsväxel i systemet betecknas som en BG (Business Group). En BG har en rad parametrar som bestämmer hur den skall bete sig, t.ex om man skall ha "samtal väntar" eller liknande tjändster inkopplade. En BG kan ha en eller flera nummerserier som betecknas "NumberSerie". Denna i sin tur har en eller flera telefonanslutningar som betecknas "Extension". Denna extension har på samma sätt som ett BG-objekt en rad parametrar som styr hur den enskilda anslutningen fungerar. För att sköta kommunikationen med servern finns en speciell klass kallad AXE, som skickar alla MML-kommandon.



Det gränssnitt jag har konstruerat är främst ett gränsnitt mot objekten "Business Group" , "Extension" och "Number serie". Dessa objekten är väldigt centrala och kan fungera som en bra referenspunkt för att vidareutveckla min lösning.

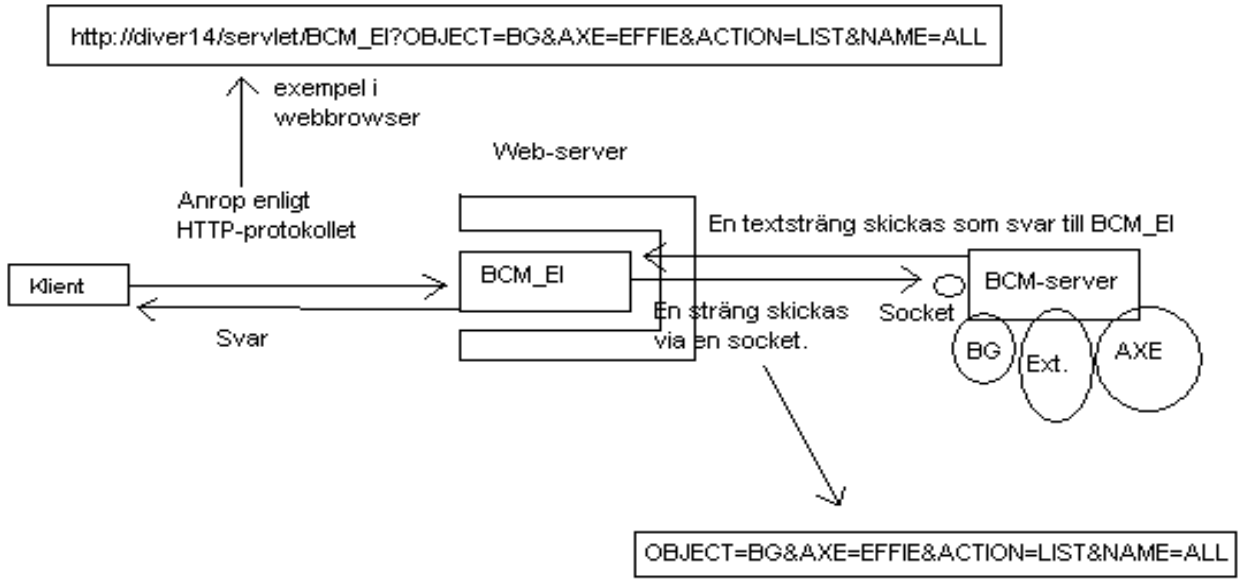
Kravspecifikation

För att lösningen skall vara tillförlitlig anser jag att det bör ställas vissa krav på systemet. Jag tänkte kort lista de krav som jag anser att systemet bör uppfylla.

1. Felhantering. En mycket central del i ett distribuerat system är felhanteringen. Denna måste vara väl utvecklad för att ett system skall kunna upplevas som trovärdigt. De möjliga felkällorna måste analyseras och lämpliga felhanteringsåtgärder skall kunna föreslås.
2. Systemet bör optimeras för prestanda.
3. Lösningen skall försöka använda sig av befintliga lösningar i nuvarande system.
4. Det skall vara möjligt att med hjälp av enkla operationer kommunicera med systemet (se även tidigare beskrivningar)
5. Tydlig beskrivning av operationer

Beskrivning av lösning

Som jag tidigare skrev har jag valt att använda mig av HTTP för att skicka anropen till BCM. Från min synvinkel är det egentligen bara intressant hur anropet skall tas emot och eftersom jag använder HTTP måste det finnas en HTTP-server, dvs en webserver, som kan ta emot anropet. Eftersom systemet redan använder sig av en webserver för kommunikationen med AXE-stationen, så kan man utnyttja en befintlig resurs.
Via en webserver startas speciella java-program, så kallade "servlets", med hjälp av ett ett HTTP-anrop. Från början hade jag tänkt köra hela systemet i webserver-miljön. Jag stötte dock på tekniska problem i samband med detta och valde istället en annan lösning. Den ser ut på följande sätt:.



System ligger på en server och lyssnar efter anrop från en klient via en socket. Servlet-programmet har jag döpt till BCM_EI . När den anropas, tar den frågesträngen och skickar den vidare till BCM-servern via en socket. BCM-servern är multitrådad (den kan köra flera processer samtidigt) och startar upp en ny tråd så fort den har läst en sträng från sin socket. Denna process omvandlar sedan strängen och utför därefter den efterfrågade operationen. När operationen är klar returneras en sträng (enligt ett fördefinierat format) till BCM_EI, som skickar den vidare till klienten.

6.4.*Beskrivning av anrop*

Som jag tidigare har beskrivit så skall systemet anropas via HTTP-protokollet. Webservern använder sig av HTTP version 1.1. Själva anrop skall vara av typen GET (se HTTP). Objektet man skall anropa är en servlet som heter BCM_EI. Modellen baseras på klient/server och innebär att klienten skickar ett HTTP-anrop, som bearbetas av servern och som sedan returnerar ett HTTP-svar.
Som jag tidigare har nämnt finns det ett antal centrala objekt i BCM-systemet och det är med utgångspunkt från dessa jag har designat gränssnittet. Varje anrop måste ha en parameter kallad OBJECT som talar om vilket objekt som operationen skall utföras på. Vidare måste anropet innehålla en parameter kallad AXE som anger namnet på den AXE-station som objektet tillhör.

Nedan följer ett exempel på hur kommunikationen via strängar kan se ut mellan klient och server.
Klient: OBJECT=BG&AXE=EFFIE&ACTION=LIST&NAME=ALL\n
Server: RESULT=SUCCESS&MESSAGE=0 OPERATION SUCCEDED\nBG_NAME=ERICSSON&CATSET=0\nBG_NAME=NODE1&CATSET=1\n
Serverns svar är utformat så att det skall vara lätt för klienten att hantera och sedan presentera för användaren. Detta var en lyckad operation, men vad händer om klienten av någon anledning inte skickar med parametern AXE.
Klient: OBJECT=BG&ACTION=LIST&NAME=ALL\n
Server: RESULT=FAILURE&MESSAGE=3 NO AXE PARAMETER\n

För en mer ingående beskrivning av min lösning hänvisar jag till bilaga 2, där en beskrivning av alla operationer med nödvändiga parametrar finns med.

7. Diskussion

Jag har nu beskrivit hur min implementation av ett kommunikationsgränsnitt till BCM ser ut och fungerar. Det kan nu vara lämpligt att återknyta till den kravspecifikation jag beskrev tidigare. Som jag ser det har jag nått ett tillfredsställande resultat på alla punkter förutom den som gäller felhanteringen. Detta beror till stor del på att HTTP-protokollet är tillståndslöst. Ett avgörande problem utgörs av att en operation oftast innebär att flera MML-kommandon skickas och så som systemet är utformat idag avbryts sändningen av MML-kommandon om ett fel uppkommer. I det befintliga systemet löses detta genom att användaren tillfrågas om det aktuella MML-kommandot skall skickas om. Detta är svårt att genomföra med min lösning, då den baserar sig på HTTP som är ett tillståndslöst protokoll. En tänkbar lösning skulle kunna vara att införa någon form av loggningsfunktion, där varje transaktion MML-kommandon skrivs till en fil med ett unikt id-nummer. Om ett fel uppkommer under sändning innehåller filen bara de MML-kommandon som har gått iväg och servern kan då skicka med ett ID-nummer för transaktionen. Klienten kan tillfråga användaren som kan välja att skicka om eller kanske få en utskrift på vilka MML-kommandon som hann sändas iväg. Om användaren väljer att utföra operationen igen, ser systemet till att endast skicka iväg de kommandon som inte finns med i loggfilen. Detta felhanteringsproblem hade dock varit lättare att lösa om ett tillståndshållande protokoll, t.ex. Corba, använts istället.

Ett annat problem med min lösning är att prestandan är lite dålig. Detta beror till stor del på att de flesta operationer innebär att flera MML-kommandon skickas och då det kan ta lång tid att få svar från ett MML-kommando innebär detta en stor tidsförlust. Detta ligger dock utanför min påverkan.

Jag har inte heller med några säkerhetsaspekter i min lösning, men jag tror att HTTP kan vara ett lämpligt protokoll eftersom det med hjälp av webservern går att lösenordskydda den servlet som anropas.

8. Slutsatser

Jag har i denna uppsats ingående studerat de tekniker, främst middlewareprodukter som kan användas för att implementera ett kommunikationsgränssnitt till BCM. Gemensamt för de flesta av dessa är att de är ganska komplexa att sätta sig in i men samtidigt väldigt användbara. Om man tittar på middlewarekategorierna jag har presenterat så anser jag att vissa av dem skulle kunna användas på BCM. Mest användbara skulle de objekt-orienterade standarderna Corba eller DCOM vara. RPC är inte tillämpligt då det endast är utformat för procedurspråk, som t.ex C. Transaktionshanterare kan inte användas på BCM för det finns ingen transaktionshantering i AXE-stationen. Däremot skulle meddelandorienterad middleware kunna tillämpas på BCM, men det skulle i princip fungera som HTTP och säkerligen vara mycket svårare att integrera med det befintliga systemet.

Jag har valt en dessa tekniker för att försöka göra en ansats på hur ett kommunikations-gränssnitt till BCM skall utformas. Med utgångspunkt från de krav som ställs på BCM så kom jag fram till att basera mitt gränssnitt på HTTP-protokollet. Den lösning jag har tillämpat fungerar och uppfyller till stor del kravspecifikationen. Dock finns vissa svagheter när det gäller felhanteringen. Valet av operationer som jag redovisar i bilaga 2 har till stor del styrts av hur det befintliga systemet var utformat. Generellt sett, om ett kommunikationsgränssnitt skall införas till ett mindre system som stödjer oberoende operationer kan det räcka med att använda ett enkelt applikationsprotokoll baserat på en standard som TCP/IP. Men i takt med att systemet kräver fler tjänster kan det vara nödvändigt att istället välja en middleware-produkt som helt enkelt har ett större utbud av tjänster. Ett kommunikationsgränssnitt till BCM ställer inte direkt några krav på sådana tjänster som middlewareprodukter kan erbjuda. Om ett större system skall byggas bör nog en middlewareprodukt användas istället.

En annan viktig anledning till att valet av kommunikationslösning föll på HTTP var att integreringen med det färdigt systemet var lätt att göra. Hade valet fallit på t.ex Corba hade flera delar av systemet behövts skrivas om.

9. Ordlista

Asynkron/förbindelselös kommunikation
En kommunikationsmodell mellan program som innebär att ingen av parterna är blockerade i väntan på svar.

BCM
Bussiness Communications Manager - ett stödsystem för att hantera virtuella företagsväxlar i AXE-stationer.

CGI
Common Gateway Interface - är ett programmeringsinterface mellan webservrar och externa program. Denna standard låter en http-klient överföra data till ett program som startas av en webserver.

FTP
Ett applikationsprotokoll som används för att föra över filer mellan datorer på internet.

MML
Man Machine Language, ett språk utvecklat av Ericsson för att kunna styra AXE-växlar. BCM-produkten innehåller de MML-kommandon som rör virtuella företagsväxlar.

Servlet
Ett java-program som körs genom en webserver. Startas genom ett HTTP-anrop.

Synkron/förbindelseorienterad kommunikation
En modell för kommunikation som innebär att klient programmet överöför data och kontroll till servern vid varje anrop och är blockerad tills ett svar returneras.

URL
Uniform Resource Locator - ett sätt att ange sökvägen till en resurs på internet.
Först anges vilket applikationsprotokoll som skall användas t.ex. http eller ftp.
Därefter anges namnet på den dator som resursen ligger på och sökvägen på den datorn till resursen. En URL kan se ut på följande sätt: http://hobbe.aom.ericsson.se/servlet/BCM_EI

10. Källförteckning

Publicerade källor

Andreasson S-A., Carlsson C., *Datakommunikation för informatik.* Chalmers Tekniska Högskola (1997)

Elbert B., Martyna B., *Client/Server Computing Architecture, Applications and Distributed Systems Management*. Norwood (1994).

Halsall F., *Data Communications, Computer Networks And Open Systems.* Addison-Wesley (1992).

Nielsen M. S., *Client/Server.* Liber Utbildning, Malmö (1995)

Internetkällor

International Systems Group, "Middleware -- The Essential Component for Enterprise Client/Server Applications", 1997 <http://www.openvms.digital.com/openvms/whitepapers/middleware/isgmidware.html>,10 april 1998

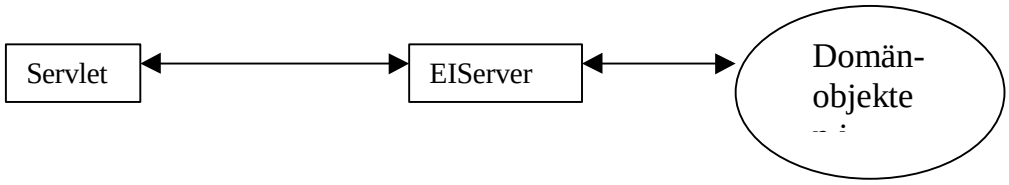
Lidfeldt Torun, "Objekten på nätet", 1998, <http://www.datateknik.se/arkiv/98-02/32.html>, 19 april 1998

Marshall James, "Easy-HTTP", 1997 <http://www.jmarshall.com/easy/http> 5 april 1998

11. BILAGA 1 (Systembeskrivning)

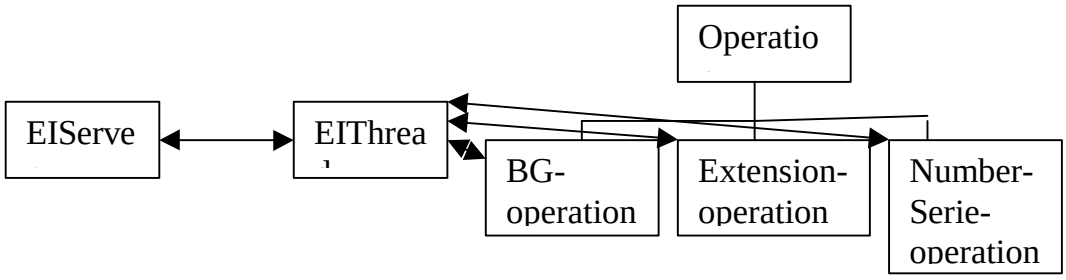
Jag har konstruerat ett gränssnitt mot BCM-system som gör att en klient via vanliga HTTP-anrop kan utbyta data med och köra operationer i systemet. Genom att kapsla in domänobjekten i den vanliga BCM-produkten och låta mitt serverprogram ta hand om anropen och ge ifrån sig svar har jag åstadkommit denna funktionalitet.

Systemet är uppbyggt på följande sätt:



Servlet-programmet består av en klass kallad BCM_EI. Denna ligger på samma dator som webservern och är den som tar emot HTTP-anrop och skickar HTTP-svar.
Om man vill anropa servleten från en webläsare så ser det ut på följande sätt.
http://192.176.46.197:9999/servlet/BCM_EI?OBJECT=BG&ACTION=LIST&AXE=EFFIE
Sökvägen till servleten är beroende av hur webservern är konfigurerad. Allt som står efter frågetecknet är indata till BCM_EI, som får detta som en sträng. BCM_EI kopplar därefter upp sig mot serverprogrammet via en socket och överför parametersträngen till servern. BCM_EI läser sedan in en sträng från samma socket och returnerar ett HTTP-svar med strängen i meddelandekroppen.

EIServern består av några olika delar som bildar ett litet system enligt följande bild. Namnen är klassbeteckningar:



Det enda EIServer gör är att lyssna på en socket efter klienter som försöker koppla upp sig. Den läser sedan in parametersträngen från sin socket, varpå den startar upp en ny tråd (process), överför parametersträngen till den tråden och återgår sedan till att lyssna efter anrop.

EIThread delar upp parametersträngen i nyckelvärdepar och använder sedan Java-språkets dynamiska klassinformation för att invokera en metod med samma namn som ACTION-parametern i en klass vars namn börjar på värdet av parametern OBJECT.
Exempel
ACTION=CREATE, OBJECT=BG
EIThread invokerar metoden CREATE() i objektet BGOperation.

De olika "Operation"-klasserna innehåller all logik för att använda domänobjekten och för att hantera fel m.m. När det gäller felhanteringen känner de till vilket nummer ett visst fel har, men inte feltexten. Denna ligger lagrad i en enkel textfil och jag har också skrivit en liten klass som läser in felmeddelanden från denna filen. Domänklasserna i sin tur innehåller all logik med MML-kommandon och sköter kommunikationen med AXE-stationen. Jag har i princip lämnat dessa klasser orörda förutom några små tillägg.

Detta var en ganska detaljerad beskrivning av hur de olika delarna i min lösning fungerar. Vill läsaren tränga ännu djupare hänvisar jag till programlistningarna i bilaga 3

12. BILAGA 2 (Operationer)

Det är så här den grundläggande kommunikationen ser ut och jag tänkte nedan beskriva de operationer som kan utföras på en BG enligt ett egendefinierat schema:

Tillåtna operationer på ett BG-objekt.

1. DELETE
2. CREATE
3. UPDATE_PROPERTY
4. UPDATE_PROPERY_TEMPLATE
5. LIST_PROPERTIES
6. LIST

DELETE

Denna operation tar permanent bort en "business group" från AXE stationen.

Nödvändiga parametrar:
AXE, BG_NAME

Felkoder:
1 NO AXE DEFINED
10 NO BG_NAME DEFINED
11 BG_NAME DOES NOT EXIST
99 AXE ERROR

Exempel:
Klient: OBJECT=BG&BG_NAME=ASICS&ACTION=DELETE
Server: RESULT=SUCCESS&MESSAGE=0 OPERATION SUCCEDED

LIST_PROPERTIES
Listar alla egenskaper som en business group har.

Nödvändiga parametrar:
BG_NAME

Felkoder
11 BG_NAME DOES NOT EXIST
99 AXE ERROR

Exempel:
Klient: OBJECT=BG&BG_NAME=KALLE&ACTION=LIST_PROPERTIES
Server: RESULT=SUCCES&MESSAGE=0 OPERATION SUCCEDED\n
p.ABEFINT=1\n
p.ACODE=2\n
osv…

CREATE
Skapar en ny Business Group. Samtidigt som en BG skapas kan man sätta dess egenskapar enligt en mall genom parametern BG_TEMPLATE.

Nödvändiga parametrar:
BG_NAME, CATSET

Valfri parameter:
BG_TEMPLATE

Felkoder
99 AXE ERROR

Exempel
Klient:
OBJECT=BG&BG_NAME=KALLE&ACTION=CREATE&CATSET=0&BG_TEMPLATE=bx.adi.---.---
Server: RESULT=SUCCES&MESSAGE=0 OPERATION SUCCEDED\n

LIST
Listar BGs

Nödvändiga parametrar:
BG_NAME

BG_NAME kan i det här fallet användas på tre olika sätt.
1. Utsökning på ett visst namn. BG_NAME=VOLVO listar en BG vid namn VOLVO
2. Utsökning på ett prefix. BG_NAME=<VO> listar de BGs som börjar på VO, exempelvis VOLVO, VOCAL osv
3. Om man vill lista alla sätts BG_NAME=ALL

Exempel
Klient: OBJECT=BG&BG_NAME=ALL&ACTION=LIST
Server: RESULT=SUCCESS&MESSAGE=0 OPERATION SUCCEDED\n
BG_NAME=HAMU&CATSET=0\n
BG_NAME=NODE&CATSET=1\n
osv…

UPDATE_PROPERTY
Uppdaterar värdet på en egenskapsparameter för en BG

Nödvändiga parametrar:
PROPERTY, PROPERTYVALUE, BG_NAME

Klient:
OBJECT=BG&BG_NAME=QTXFRMA&ACTION=UPDATE_PROPERTY&PROPERTY=ABEFINT&PROPERTYVALUE=1
Server: RESULT=SUCCESS&MESSAGE=0 OPERATION SUCCEDED\n

UPDATE_PROPERTY_TEMPLATE
Uppdaterar flera egenskapsparametrar på en gång genom att ange en mall.

Nödvändiga parametrar:
PROPERTY_TEMPLATE, BG_NAME

Felkoder
22 Wrong property template
99 AXE error

Klient:
OBJECT=BG&BG_NAME=QTXFRMA&ACTION=UPDATE_PROPERTY_TEMPLATE&PROPERTY_TEMPLATE=bx.---.edu.abe
Server: RESULT=SUCCESS&MESSAGE=0 OPERATION SUCCEDED\n

13. BILAGA 3 (Programlistningar)

Klassen BGOperation

```
import java.io.*;
import java.net.*;
import java.util.Properties;
import lme.nisce_domain.*;
import lme.util.Association;
import lme.util.Lme;
import lme.util.ManagementException;
import lme.util.PropertyType;
import java.lang.StringBuffer;
import java.util.Enumeration;
import lme.nisce_ui.Templates;

class BGOperation extends Operation {

	String		_error;
	String		_name;
	String		_version;
	String		_axe;
	String		_action;
	String		_BGproperty;
	String		_propValue;
	Association[] _nes;
	AXE			_ne;
	BG			_bg;

	String getFirst()
	{	_axe = _param.getProperty("AXE");
		_name = _param.getProperty("NAME");
		if ( _axe==null)
			return _failure+"2 "+_errorReader.getMessage(2);
		if (_name==null)
			return _failure+"3 "+_errorReader.getMessage(3);
		return null;
	}

	public String CREATE()
	{
		_error=getFirst();
		if (_error != null)
			return _error;
		return createBG(_axe,_name);

	}

	public String DELETE()
	{
	_error=getFirst();
```

```
        if (_error != null)
                return _error;

return deleteBG(_axe,_name);
 }


 public String UPDATE()
 {
 _error=getFirst();
        if (_error != null)
                return _error;
        return update(_axe,_name);
 }

 public String LIST() {
 _axe = _param.getProperty("AXE");
 if (_axe==null)
                return _failure+"2"+_errorReader.getMessage(2);
        return listBG(_axe);
 }

 public String UPDATE_PROPERTY_TEMPLATE()
 {       _error=getFirst();
        if (_error != null)
                return _error;
        return update_template(_axe,_name);
 }

 public String LIST_PROPERTIES()
 {
        _error=getFirst();
        if (_error != null)
                return _error;
        return list_properties(_axe,_name);
 }
 public BG[] listBGs(String name) throws ManagementException
 {

                        AXE axe = getAXE(name);
                        if ( axe != null ) {
                                BG tmpBGs[] = (BG[]) axe.list( null, "BG" );
                                return tmpBGs;
                        }
                        return null;
 }

 String createBG(String neName, String bgName)
 {
        _ne = getAXE(neName);
        if ( _ne != null ) {
                _bg = (BG) _ne.template("BG");
                _bg.setName(bgName);
                try
```

```
            {
                _bg.agent().create(_bg);
            } catch (ManagementException me)
            {       return "FAILURE\nMESSAGE:99";}
            return "SUCESS\nMESSAGE:0";

        }
        return "FAILURE\nMESSAGE:2";
    }


String deleteBG(String neName, String bgName){
    _ne = getAXE(neName);
        if ( _ne != null ) {
            BG bg = null;
            try
            {
                bg = getBG(neName, bgName);
            } catch (ManagementException me)
              {
                    String nr=getErrorNumber(me);
                    return _failure+
                        nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                        "\n"+"PRINTOUT="+
                        me.getErrorText()+"\n";
              }

                if (bg != null ) {
                    try
                    {
                        bg.agent().delete(bg);
                    } catch (ManagementException me)
                     {
                        String nr=getErrorNumber(me);
                        return _failure+
                                nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                                "\n"+"PRINTOUT="+
                                me.getErrorText()+"\n";
                    }

                return _success;
            }
            return "FAILURE\nMESSAGE:6";
    }
        return _failure+"9 "+_errorReader.getMessage(9);

    }


    String update(String neName, String bgName) {
        String propValue;
        String propName;
        BG bg=null;
        try
```

```
        {
              bg = getBG(neName, bgName);
        } catch (ManagementException me)
          {     String nr=getErrorNumber(me);
                    return _failure+
                          nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                          "\n"+"PRINTOUT="+
                          me.getErrorText()+"\n";
          }

    propName=_param.getProperty("PROPERTY");
    if (propName == null)
          return _failure+"12 "+_errorReader.getMessage(12)+"\n";
    propValue = _param.getProperty("PROPERTYVALUE");
    if (propValue==null)
          return _failure+"13 "+_errorReader.getMessage(13)+"\n";
    bg.put("p." + propName,propValue);
    try
          {
              bg.agent().update(bg);
              return _success;
          } catch (ManagementException me)
                    {
                          String nr=getErrorNumber(me);
                              return _failure+
                                        nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                                        "\n"+"PRINTOUT="+
                                        me.getErrorText()+"\n";
                    }
    }

String listBG(String neName) {
    try
    {
    String bgName=_param.getProperty("NAME");
    if (bgName == null)
          return _failure+"4 "+_errorReader.getMessage(4)+"\n";;
    BG[] bgs=listBGs(neName);
    StringBuffer result = new StringBuffer();
    if (bgs == null)
          return _failure+"5 "+_errorReader.getMessage(5)+"\n";

    result.append(_success);

    if (bgName.substring(0,1).equals("<")
        && bgName.substring(bgName.length()-1).equals(">") )
    {
          String prefix = bgName.substring(1, bgName.length()-1 );

          for (int i=0; i< bgs.length;i++)
          {
              if ( (bgs[i].name()).startsWith(prefix) )
              {
                    result.append("BG_NAME="+bgs[i].name()+"&"+
```

```
                                "Catset="+String.valueOf(bgs[i].catset())+
                                "\n" );
                        }
                }
        } else
        {
                for (int i=0; i< bgs.length;i++)
                {
                        if (bgName.equals(bgs[i].name()) || bgName.equals("ALL") )
                        {
                                result.append("BG_NAME="+bgs[i].name()+"&"+
                                        "Catset="+String.valueOf(bgs[i].catset())
                                        +"\n");
                        }
                }
        }
        return new String(result);
        } catch (ManagementException me)
                {
                        String nr=getErrorNumber(me);
                                return _failure+
                                        nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                                        "\n"+"PRINTOUT="+
                                        me.getErrorText()+"\n";
                }

}

String update_template(String neName, String bgName)
{       Properties bgProperties;
        BG bg=null;
        try
                {
                        bg = getBG(neName, bgName);
                } catch (ManagementException me)
                 {
                        String nr=getErrorNumber(me);
                                return _failure+
                                        nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                                        "\n"+"PRINTOUT="+
                                        me.getErrorText()+"\n";
                 }
        String template = _param.getProperty("BG_TEMPLATE");
        bgProperties = Templates.Load(template);
        if (bgProperties==null)
                return _failure+"22 "+_errorReader.getMessage(22)+"\n";
        bg.put(bgProperties);
        try
                {
                        bg.agent().update(bg);
                        return _success;
                } catch (ManagementException me)
                        {
                                String nr=getErrorNumber(me);
```

```
                        return _failure+
                            nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                            "\n"+"PRINTOUT="+
                            me.getErrorText()+"\n";
                    }
    }


    String list_properties(String neName, String bgName)
    {
        BG bg=null;
        try
        {
            bg = getBG(neName, bgName);
            AXE axe = getAXE(neName);
            bg.setName(bgName);
            bg.setAgent(axe);
            PropertyType[] pt = bg.propertyTypes("p.");
            bg.agent().get(bg,pt);
            Properties p = bg.properties("p.");
            StringBuffer result = new StringBuffer();
            result.append(_success);
            for (Enumeration e= p.propertyNames();e.hasMoreElements();)
            {
                String temp = (String) e.nextElement();
                result.append( temp+"="+p.getProperty(temp)+"\n" );
            }
            return new String(result);

        } catch (ManagementException me)
            {
                String nr=getErrorNumber(me);
                    return _failure+
                        nr+" "+_errorReader.getMessage(Integer.parseInt(nr))+
                        "\n"+"PRINTOUT="+
                        me.getErrorText()+"\n";
            }


    }

    String run() {
        return null;
    }

}
```

Klassen EIServer

```
package lme.bcm_ei;
import java.net.*;
import java.io.*;

public class EIServer {
   public static void main(String[] args) throws IOException {
      ServerSocket serverSocket = null;
      boolean listening = true;

      try {
         serverSocket = new ServerSocket(4444);
      } catch (IOException e) {
         System.err.println("Could not listen on port: 4444.");
         System.exit(-1);
      }

      while (listening)
         new EIThread(serverSocket.accept()).start();

      serverSocket.close();
   }
}
```

Klassen EIThread

```
package lme.bcm_ei;

import java.util.StringTokenizer;
import java.util.Properties;
import java.lang.*;
import java.lang.reflect.*;
import java.util.NoSuchElementException;
import java.net.*;
import java.io.*;

public class EIThread extends Thread {
      String _operation;
      Properties list;
      Socket _answerSocket;
      PrintWriter _out;
      boolean _isError=false;

     public EIThread(Socket sock) {
               if (sock == null)
               {
                     System.out.println("Could not get client socket");
                     _isError=true;
                     return;
```

```
                }
                _answerSocket = sock;

                try
                {
                        _out = new PrintWriter(_answerSocket.getOutputStream(),true);
                } catch (IOException e)
                 {
                        System.out.println("Client must be down");
                        _isError=true;
                 }

        }

                void setOperation(String op) {

                        if ( op.equals("null")) {

                                        _out.println("RESULT=FAILURE&MESSAGE=99&PRINTOUT=NO
QUERYSTRING\n");
                                        _isError=true;
                                        try
                                        {
                                                _answerSocket.close();
                                        } catch (IOException e) {e.printStackTrace();}
                                        return;

                        }
                        _operation = op.trim();
                }


        public void run()
        {

                if (_isError==true)
                {

                        return;
                }


                        BufferedReader br=null;

                  try {
                        _out = new PrintWriter(_answerSocket.getOutputStream(),true);
                         br= new BufferedReader(
                                                new InputStreamReader(
                                                _answerSocket.getInputStream()));

                        String input = br.readLine();
                        setOperation(input);
                        } catch (IOException e) { return; }
                        if (_isError)
```

```
                {

                        return;
                }

        StringTokenizer cmdLine;
        list = new Properties();

        StringTokenizer st = new StringTokenizer(_operation,"&");
        if (!st.hasMoreTokens()) {
                        _out.println("RESULT=FAILURE&MESSAGE="
                                        +"97&PRINTOUTS=ERROR IN"
                                          +"THE COMMANDLINE");
                        try
                        {
                                _answerSocket.close();
                        } catch (IOException e) {e.printStackTrace();}
                        return;
        }
          while (st.hasMoreTokens()) {
                 String field = st.nextToken();
                  cmdLine = new StringTokenizer(field,"=");
                  if (cmdLine.hasMoreTokens()) {

                        while (cmdLine.hasMoreTokens()) {
                                try {
                                  list.put(cmdLine.nextToken(),cmdLine.nextToken());
                                } catch (NoSuchElementException e) {
                                        _out.println("RESULT=FAILURE&MESSAGE="
                                                                                        +"97&PRINTOUTS=ERROR IN"
                                                                                          +"THE COMMANDLINE");


                                        return;
                                 }
                        }
                                        }
          }

          list.list(System.out);
          String obj = list.getProperty("OBJECT");
                        if (obj==null)
                        {
                                _out.println("RESULT=FAILURE&MESSAGE=98&PRINTOUT=NO OBJECT
DEFINED");
                                return;
                        }
          String action = list.getProperty("ACTION");
          if (action==null)
                        {
                                _out.println("RESULT=FAILURE&MESSAGE=98&PRINTOUT=NO ACTION
DEFINED");
                                return;
                        }
```

```
                    try {
                    Class opClass = Class.forName("lme.bcm_ei." + obj + "Operation");
                    Operation op = (Operation) opClass.newInstance();
                        op.setFields(list);
                        Method m = opClass.getMethod( action, null  );
                        Object report = m.invoke(op,null);
                        _out.println(report);
                        _answerSocket.close();

                }
            catch( InstantiationException e ) {
                    _out.println(("RESULT=FAILURE&MESSAGE=99&PRINTOUT=INTERNAL ERROR"));
                return;
            }
            catch( ClassNotFoundException e ) {
            _out.println( ("RESULT=FAILURE&MESSAGE=98&PRINTOUT=NOT A VALID OBJECT"));
            return;
            }
            catch( IllegalAccessException e ) {
            System.out.println( "Access problems " + e); //shouldn't happen
            return;
            }
                catch( InvocationTargetException e ) {
                    e.getTargetException().printStackTrace();
                }
                catch( NoSuchMethodException e ) {
                    _out.println( ("RESULT=FAILURE&MESSAGE=98&PRINTOUT=NOT A VALID
ACTION"));
                }
                catch( Exception e ) {
                    e.printStackTrace();
                }


}

}
```

Klassen MessageReader

```
package lme.bcm_ei;

import java.io.*;
import java.util.StringTokenizer;

public class MessageReader
{
	String		_path;

	public MessageReader() {}

	public void setPath(String aPath)
	{
		if (aPath != null)
			_path=aPath;
	}

	public String getMessage(int messageNumber)
	{
		return readFile(messageNumber);
	}

	String readFile(int key)
	{
		StringTokenizer		st;
		String			text;
		String			input;
		String			number;
		try
		{
			BufferedReader br = new BufferedReader(new FileReader
								("printouts.config"));
			while ( (input=br.readLine()) != null)
			{
				st = new StringTokenizer(input,"=");
				if (st.hasMoreTokens())
				{
					number = st.nextToken();
					if (number.equals(String.valueOf(key) ) )
					{
						text =  st.nextToken();
						br.close();
						return text;
					}
				}
			}
			return null;
		} catch (IOException e) { e.printStackTrace();}
```

```
        return null;
    }

    public static void main(String args[])
    {
        System.out.println((new MessageReader()).getMessage(1));
    }
}
```

Klassen Operation

```
package lme.bcm_ei;

import java.util.Properties;
import java.net.*;
import java.util.Properties;
import lme.nisce_domain.*;
import lme.util.Association;
import lme.util.Lme;
import lme.util.ManagementException;

public class Operation {
    Properties      _param;
    String          _success        = "RESULT=SUCCES&MESSAGE=0 OPERATION SUCCEDED\n";
    String          _failure        = "RESULT=FAILURE&MESSAGE=";
    MessageReader   _errorReader    = new MessageReader();

    public void setFields(Properties list) {
        _param = list;
        String      server  = "diver14";
        int         port    = 9000;
        Lme.TraceLevel = Lme.SEVERE;

        try {

            AXE.SetServerURL( new URL(
                    "http://" + server + ":"+ port + "/server-java/" ) );
            AXE.SetCodeBase(new URL("file:/home/scuba/qtxfrma/lme/nisce_ui/"));
        } catch (MalformedURLException e) {System.out.println("Fel URL!");}
    }

    BG getBG(String neName, String bgName ) throws ManagementException
    {
        return (BG) getAXE(neName).get(bgName,"BG");
    }

    AXE getAXE(String name)
    {
        String key = name;
        return AXE.getAXE(name);

    }
```

```
    String getErrorNumber(ManagementException me)
    {

        if (me.getFaultType()==0)
            return "98";
        if (me.getFaultType()==1)
            return "99";
        return "100";
    }

}
```

Textfilen med felkoder

0=OPERATION SUCCEDED
1=INTERNAL SERVER ERROR
2=NO AXE
3=NO BG
4=WRONG BG NAME
5=COULD NOT LIST BG GROUPS
9=AXE DOES NOT EXIST
12=NO PROPERTY DEFINED
13=NO PROPERTYVALUE
22=WRONG TEMPLATE
98=ERROR WHEN SENDING TO THE AXE
99=ERROR IN THE AXE