Representational State Transfer för kommunikation i gridsystem

Stefan Toivonen

Kandidatavhandling i datavetenskap

Handledare: Marina Waldén

Tekniska Fakulteten

Åbo Akademi

2011

Referat

Sökord: gridteknik, tjänste-orienterad arkitektur, representational state transfer

Innehåll

1. Inledning.. 1
2. Gridteknik.. 1
3. Tjänsteorienterad arkitektur.. 2
3.1 Översikt.. 2
3.2 Principer.. 3
3.3 Användning..4
4. Representational State Transfer..5
4.1 Arkitektonisk översikt.. 5
4.1.1 Klient-server.. 5
4.1.1 Tillståndslöshet.. 6
4.1.2 Cache.. 6
4.1.3 Enhetligt gränssnitt.. 7
4.1.4 Lager-på-lager system..8
4.1.5 Code-on-demand..8
4.2 Arkitektoniska element..9
4.2.1 Dataelement.. 9
4.2.1.1 Resurser och resursidentifierare..11
4.2.1.2 Representationer..12
4.2.2 Sammankopplare.. 14
4.2.3 Komponenter.. 17
4.3 Användning..19
5. Sammanfattning.. 20
Referenser.. 20

1. Inledning

Målsättning.

2. Gridteknik

Ett gridsystem är ett distribuerat system som har som mål att skapa en illusion av en stor och kraftig virtuell dator. Gridsystem består av en stor samling av sammankopplade heterogena system som delar på en varierande uppsättning av resurser. Ett gridsystemet koordinerar de distribuerade resurserna genom att använda sig av standardiserade och öppna samt generella protokoll och ett gränssnitt som skall ge en hög kvalitetsgaranti.

Ett gridsystem integrerar och koordinerar resurser och användare som finns inom flera olika domäner. Den sköter om saker som gäller systemets säkerhet, politik, medlemskap, etc. Utan denna kapabilitet så skulle det endast vara ett lokalt system.

Gridsystem byggs upp med hjälp av olika protokoll och gränssnitt som har flera olika användningsområden. De adresserar saker som autentisering, auktorisering, upptäckande av resurser och tillgång till resurser. Det är mycket viktigt att dessa protokoll och gränssnitt är standardiserade och öppna för annars blir de applikationsspecifika.

En hög kvalitetsgaranti skall göra det möjligt att använda sig av resurser på ett koordinerat sätt. Kvalitetsgarantin relaterar till saker som responstid, överföringshastiget, tillgänglighet och säkerhet. Även delandet av olika typer av resurser för att möta de varierande behov som användarna har.

3. Tjänsteorienterad arkitektur

3.1 Översikt

Allmänt kan man säga att en tjänsteorienterad arkitektur (eng. service-oriented architecture, SOA) är en stil för att göra det möjligt att bättre integrera IT-system så att de tjänar ett företags behov. En tjänsteoriented arkitektur kan ses ur två olika perspektiv. Den kan ses ur ett företagsperspektiv eller ur ett tekniskt perspektiv.

Ur företagsperspektivet så kan man tänka sig en tjänsteorienterad arkitektur som en mängd av dynamiska tjänster och processer som de erbjuder åt sina kunder, samarbetspartners eller internt inom företaget. På det sättet kan tjänsterna kombineras och kompletteras för att bättre stöda ändring eller utveckling i ett företags behov och modeller.

Ur ett tekniskt perspektiv så kan man se en tjänstorienterad arkitektur som något som definierar mjukvara i begrepp som tjänster. Mjukvaran är sedan implementerade i komponenter som kan kallas på för att utföra en företagsuppgift. Det finns flera sätt att beskriva vad en tjänsteorienterad arkitektur är och vad den har för funktion ur en teknisk synpunkt.

Organization for the Advancement of Structured Information Standards (OASIS) definierar en tjänsteorienterade arkitektur som en sätt att organisera och använda sig av distribuerade resurser. Dessa distribuerade resurser kan vara under kontroll av flera domäner som i sin tur kan ha olika ägare.

Resurser skapas av entiteter, det vill säga människor och organisationer, för att lösa eller ge stöd åt ett problem som de försöker lösa i företagets dagliga görande. Man kan tänka det som att en persons behov tillfredsställs av resurser av vad en annan person erbjuder. I ett gridsystem så kan man tänka det som att en dators krav tillfredsställs av en dator som hör till någon annan.

Det behöver inte alltid vara så att behovena och resurserna har ett ett-till-ett samband. Granulariteten kan variera från något mycket simpelt till något mycket komplext. Ett behov kan kräva en kombination av flera resurser eller en resurs kan adressera flera än endast ett behov.

3.2 Principer

Tjänster kan beskrivas som mjukvarukomponenter som har väldefinierade gränssnitt och inte är bunda till någon specifik implementation. En viktig sak är att tjänstens gränssnitt, som säger vad den skall göra, och implementation, som säger hur den skall göra det, är helt separata. På detta vis så behöver klienterna, som utnyttjar tjänsterna, inte veta något om hur tjänsterna utför deras förfrågningar.

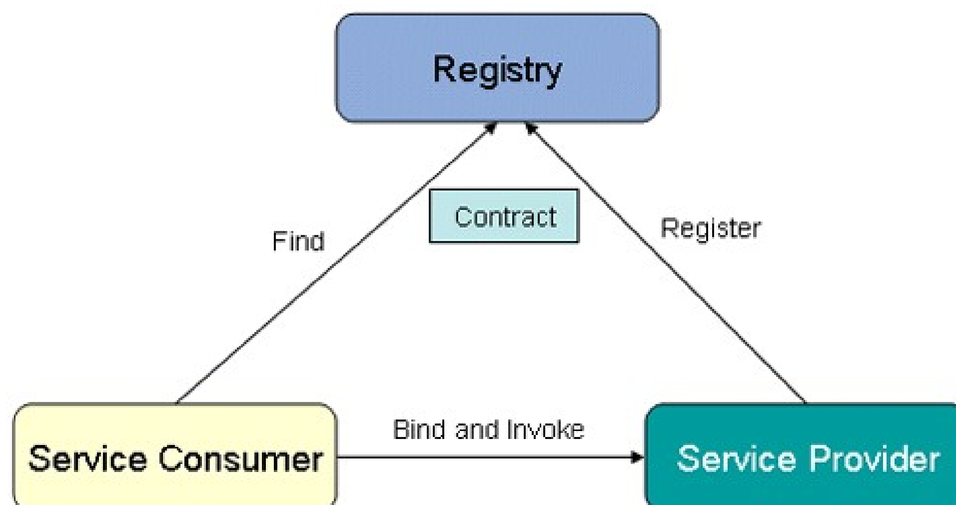
Tjänster följer den principen att de skall vara self-contained, som betyder det att de utför fastställda funktioner och behöver inte använda sig av externa tjänster.

Tjänsterna skall också vara löst kopplade med varandra. Det betyder att de skall ha, eller använda sig av, så lite information om de andra tjänsterna som möjligt. I ett optimalt fall så behöver de inte någon information alls om andra tjänsterna. På grund av den lösa koppling så blir tjänsterna självständiga och behöver inte stöd av någon annan tjänst för att fungera. Det betyder också att de har inget behov av data från varandra. Varje löst kopplad miljö upprätthåller en kopia (eller en delmängd) av data och tjänster. I en löst kopplad miljö så är redundans en god design inte en dålig design.

Tjänsterna skall också vara möjliga att hittas på ett dynamiskt sätt. Detta gör det möjligt att hitta tjänster utan någon direkt inblanding av en användare. På det sättet går det också enkelt att introducera nya tjänster eller att ta bort sådana som inte mera används.

Det skall också vara möjligt att skapa komposita tjänster. De kan byggas upp genom att kombinera olika tjänster med varandra.

En tjänsteorienterad arkitektur använder sig av find-bind-execute paradigmen, se Figur 1. I denna paradigm så registrerar de som erbjuder tjänsterna sina tjänster i ett allmänt register. Detta register används sedan av konsumenterna för att hitta en tjänst som överenstämmer med deras behov. Om registret har en tjänst som passar konsumenten så skapas ett kontrakt och en ändpunktsadress till den ifrågavarande tjänsten.



Figur 1: Find-bind-execute paradigm

Applikationer som är baserade på en tjänsteorienterad arkitektur delas i flera lager som presentations-, företagslogiks- och persistancelager. Tjänsterna är de byggnadsblock som används för att bygga upp en applikation som baserar sig på en tjänsterorienterad arkitektur. Det är möjligt att implementera vilken funktion som helst som en tjänst men problemet är att definiera ett gränssnitt som är på den rätta abstraktionsnivån. Tjänsterna skall ha en grovkorning funktionalitet. Med det menar man att funktionaliteten inte skall vara uppdelad i allt för många och små beståndsdelar.

3.3 Användning

På grund av att en tjänsteorienterad arkitektur inte är bunden till någon specifik implementation så går det att använda sig av flera olika teknologier för att realisera den. Några exempel på teknologier som kan användas vid implementering så kan man nämna: SOAP, Remote Procedure Call (RPC), Representational State Transfer (REST), Distributed Component Modell (DCOM), Common Object Request Broker Architecture (CORBA), Web Services och Data Distribution Service (DDS). Det går att använda sig av en eller flera av dessa teknologier vid implenteringen av en tjänsteorienterad arkitektur.

4. Representational State Transfer

Representational State Transfer (REST) är en arkitektonisk stil ämnad för distribuerade hypermedia system. Hypermedia är en logisk förlängning av termen hypertext där grafik, audio, video, vanlig text och hyperlänkar bygger upp ett icke-linjärt medium av information. En arkitektonisk stil i sin tur innebär att det inte är en konkret systemarkitektur utan en mängd av begränsningar som används vid designadet av en systemarkitektur. Som en jämförelse kan man ta Louvre i Paris som är en konkret arkitektur av en arkitektonisk stil, barockstilen.

4.1 Arkitektonisk översikt

REST är en hybrid arkitektonisk stil som har påverkats av flera olika nätverks-baserade arkitektoniska stilar men med skillnanden att den har ett enhetligt gränssnitt. I designadet så har man utgått från ett system som är helt utan några begränsningar. Sedan har systemets olika element identifieras och deras beteenden har begränsats så att de överensstämmer med de krav som ställs. Designen har sedan fortsatt på ett evolverande sätt så att den till slut återspeglar vad som väntas av en modern webbarkitektur. REST betonar saker som skalbarheten av komponenternas interaktioner, allmänhet när det gäller gränssnitten, självständiga komponenter samt användet av förmedlande komponenter.

4.1.1 Klient-server

Den första begränsningen som har gjorts är att den följer klient-server arkitektoniska stilen. En server erbjuder en mängd av olika tjänster och lyssnar på inkommande förfrågningar. En klient skickar iväg förfrågningar till en server vilkas tjänster de vill använda sig av. Servern väljer att förkasta förfrågningen eller processera den genom att skicka ett svar tillbaka till klienten.

Orsaken till valet av denna arkitektoniska stil är att det tillåter att man delar upp begräsningarna. Genom att separera användargränssnittet från lagringen av data så ökar man portabiliteten. Skalbarheten förbättras genom att servern består av

simpla komponenter.

4.1.1 Tillståndslöshet

En begränsning som läggs till är det att kommunikationen skall vara tillståndslös (eng. stateless). Det betyder att varje förfrågning från en klienten till servern måste innehålla all den information som behövs för att förfrågningen skall kunna tolkas korrekt. Föfrågningen får inte använda sig av någon kontextuell information som har sparats på servern. Upprätthållningen av sessionstillståndet är enbart upp till klienten.

Denna begränsning ger upphov till egenskaper som visibilitet, pålitlighet och skalbarhet. Med visibilitet så menas en komponents förmåga att övervaka och förmedla interaktioner mellan två komponenter. Visibiliteten förbättras genom att övervakande system inte behöver se på mera än en förfrågning för att bestämma innebörden av förfrågningen. Pålitligheten förbättras för att det underlättar arbetet att återhämta sig från partiella olyckor. Skalbarheten förbättras på grund av att man inte behöver spara information om tillståndet mellan förfrågningar. Detta tillåter serverkomponenterna att effektivt frigöra resurser och simplifierar implementeringen. Implementering blir simplare för att servern inte behöver sköta om användingen av resurser från en förfrågning till en annan.

Tillståndslösheten har också vissa nackdelar. En nackdel är att den kan minska på nätverkets effektivitet genom att mängden av repetitiv data som skickas i en serie av förfrågningar ökar. Orsaken till detta är att data inte kan lagras på servern i en delad kontext. En annan nackdel är att på grund av att tillståndet av applikationerna sköts av klienten så försämras serverns kontroll över att applikationerna hålls konsekventa.

4.1.2 Cache

För att förbättra effektiviteten av nätverket så skall man använda sig av en cache. Av ett svar så krävs det att det implicit eller explicit markeras om det är möjligt att

lagra svaret i en cache eller inte. Om det är möjligt att lagra svaret så får klientens cache rättigheten att använda svaret för senare, men ekvivalenta, förfrågningar. Fördelen med en cache är att det är möjligt att delvis eller totalt eliminera vissa interaktioner och att effektiviteten förbättras samt skalbarheten. Användarsynen på nätverkets prestationen förbättras genom att reducera svarstiden i en serie av interaktioner. Nackdelen är att en cache kan reducera pålitligheten om föråldrad data inom cachen skiljer sig mycket från datan som skulle ha erhållits om förfrågningen hade gått direkt till servern.

4.1.3 Enhetligt gränssnitt

En stor tyngdpunkt i REST ligger på ett enhetligt gränssnitt (eng. uniform interface) mellan olika komponenter. Genom att ha ett allmänt komponentgränssnitt så blir systemarkitekturen enklare och synbarheten av interaktionerna förbättras. För att göra det möjligt för tjänsterna att utvecklas stjälvständigt så är implementationen inte kopplade till de tjänster som de erbjuder. Nackdelen är att ett enhetligt gränssnitt försämrar effektiviteten. Orsaken är att informationen överförs i en standardiserad form, istället för i en form som är skräddarsydd för applikationens behov. REST gränssnittet är designat att vara effektivt för grovkorniga (eng. large-grain) hypermedia dataöverföringar. REST och HTTP fungerar på samma sätt genom att ha ett begränsat antal av metoder för att utföra alla möjliga operationer på en resurs. I HTTP så används GET för att hämta en representation, PUT ersätter data eller skapar en tom resurs, POST lägger till en ny resurs då en ny URI måste skapas annars så utökas resursen som redan existerar, DELETE tar bort en resurs och HEAD/OPTIONS så hämtar metadatan av en resurs.

Gränssnittet i REST definieras av fyra begränsningar: identifikation av resurser, manipulation av resurser genom representation, självbeskrivande meddelande, hypermedia som motor för applikationstillstånd. Identifikation av resurser sker genom användningen av Uniform Resource Identifiers (URI). Med manipulation av resurser genom representation menas det att resurserna inte direkt manipuleras eller accesseras utan endast deras representationer. Med självbeskrivande

meddelanden menar man att resursen är inte kopplad till sin representation så att innehållet kan kommas åt i olika format (t.ex. HTML, XML, PDF, JPEG, etc). Hypermedia som motor för applikationstillstånd gör det möjligt att en applikations tillståndet hålls i ett eller flera dokument, som finns antingen på klienten eller på servern.

4.1.4 Lager-på-lager system

Lager-på-lagersystemet (eng. layered system) gör det möjligt för en arkitektur att bestå av olika hierarkiska lager genom att begränsa komponenternas beteende. På det sättet kan man begränsa att varje komponent inte kan se längre än de omedelbara lagren de interagerar med. Det gör att man begränsar systemet komplexitet genom att begränsa kunskapen av systemet till ett lager. Lagrena kan användas för att enkapsulera legacy-tjänster och för att skydda nya tjänster från legacy-klienter. Genom att flytta den sällan utnyttjade funktionaliteten till en delad förmedlare så gör man komponenterna simplare. Förmedlare kan också användas för att förbättra systemets skalbarhet. De gör det även möjligt att balansera tjänsternas bördor (eng. load balancing) över flera nätverk och processorer.

En stor nackdel med ett lager-på-lager system är att det skapas mera overhead och det leder till en ökning i svarstiden vid processering av data. Ur användarens synpunkt så leder detta till en reduktion i effektiviteten men detta kan motarbetas genom att använda delade cachen vid förmedlarna.

Förmedlande komponenter kan också aktivt ändra på innehållet av meddelande medan de överförs. Orsaken är att meddelandena är självbeskrivande och deras innebörd är synlig och kan tolkas av förmedlarna.

4.1.5 Code-on-demand

En klients funktionalitet kan utökas genom att använda sig av appletprogram och skript. Det gör det möjligt att ladda ner programkod som kan exekveras av klienten. På det sättet minskar behovet av att ha en stor del av funktionalitet

färdigt implementerad. Det tillåter att ny funktionalitet laddas ner efter att klienten har blivit installerad och på det sättet ökar man på systemets egenskaper. Samtidigt orsakar man en reduktion av synbarheten och det är orsaken till att det är en valbar begränsning.

Det betyder att systemet endast njuter av fördelarna (och nackdelarna) när den är i användning i ett visst område i systemet. Ett exempel är om klientmjukvaran inom en organisation stöder appletprogram i Java så kan tjänster byggas så att fördelarna av funktionaliten av nedladdbara Java klasser utnyttjas. Samtidigt kan organisationens brandmur förneka överförandet av Java applets från externa källor och från utsidan ser systemet ut att inte ha stöd för code-on-demand. En valbar begränsning gör det möjligt att designa en arkitektur så att den i det allmänna fallet stöder en egenskap men samtidigt förbjuds i ett specifikt fall.

4.2 Arkitektoniska element

Det går att dela in de arkitektoniska elementen i tre olika klasser: dataelement, komponenter (components), sammankopplande element (connectors). REST bryr sig inte om hur komponenterna implementeras eller syntaxen av protokollen. Istället ligger tyngdpunkten på komponenternas roller, begränsningarna som påverkar deras interaktioner och hur de tolkar betydelsefulla element. Detta bestämmer hur begränsningarn påverkar komponenterna, sammankopplarna och datan och hur den fungerar som en nätverksbaserad applikation.

4.2.1 Dataelement

Ett dataelement är ett element av information som överförs eller tas emot av en komponent genom att använda sig av en sammankopplare. I andra arkitektoniska stilar så är all data enkapsulerad och gömd men i REST så är innebörden och tillståndet av datan en viktig del. Orsaken till denna design kan ses i egenskaperna för ett distribuerad hypermedia system. Vid valet av en länk så måste informationen flyttas från platsen den lagras till platsen där den skall användas. Denna process skiljer sig från många andra sätt där det oftast är enklare och

effektivare att flytta den processerande agenten (t.ex. mobilkod, sökuttryck) till datan, istället för att flytta datan till den som skall processera det.

I en distribuerad hypermediaarkitektur kan man lösa detta problem på tre olika sätt. Det första sättet är att datan skapas på stället där den är lagrad. Sedan skickas datan i ett fastställt format över till mottagaren. Det andra sättet är att enkapsulera datan, med hjälp av en renderingsmotor, och skicka båda över till mottagaren. Det tredje sättet är att skicka data som innehåller metadata över till mottagaren. Metadatan beskriver typen av data som överförs. På detta sätt så kan mottagaren själv bestämma vad för renderingsmotor de vill använda sig av. En renderingsmotor, ur en webbsynpunkt, är en mjukvarukomponent som tar ett märkspråk (t.ex. HTML, XML, bilder) och formaterad information (t.ex. CSS, XSL) för att sedan visa det formaterade materialet på skärmen.

Varje av dessa tre sätt att lösa problemet, överföring av data, har sina fördelar och nackdelar. Den första sättet tillåter att all information om datan hålls gömd. Det förhindrar att man gör antaganden som gäller datan. Detta följer den traditionella klient-server modellen och fördelen är att det underlättar implementeringen av klientprogramvaran. Nackdelen är att det strängt begränsar mottagarens funktionalitetet och placerar en stor del av processeringsbördan på sändaren, som orsakar problem med skalbarheten. Det andra sättet tillåter gömmande av information och samtidigt tillåter en specialiserad processering av datan genom att använda en renderingsmotor. Detta begränsar funktionaliteten av mottagaren och kan även drastiskt öka mängden av data som måste överföras. Den tredje möjligheten tillåter att sändaren förblir enkel och skalbar och samtidigt att mängden av överföring hålls på en låg nivå. Det som går förlorat är fördelen med gömmandet av information och kräver att både sändaren och mottagaren förstår sig på samma datatyper.

REST är en kombination de olika sättena. Den fokuserar på en delad förståelse av datatyper med hjälp av metadata, men begränsar omfattningen av vad som visas av det allmänna gränssnittet. Komponenterna i REST kommunicerar med varandra genom att överföra en representation av datan. Om representationen är i

samma format eller härledd från källan göms av gränssnittet. På det sättet så undviker man de portabilitet- och skalbarhetsproblem som har med klient-server stilen att göra. Det allmänna gränssnittet gör det möjligt att gömma information och som gör det möjligt att utnyttja sig av enkapsulering.

Data element	**Moderna på Webben**
Resurs	Det konceptuella målet av en hypertext referens
Resursidentifierare	URL, URN
Representation	HTML dokument, JPEG bild
Representation metadata	Mediatyp, tiden när den senast blev modifierad
Resurs metadata	Källänken
Kontrolldata	Cache kontroll

Lista 1: REST dataelement

4.2.1.1 Resurser och resursidentifierare

All information som kan ges ett namn kan vara en resurs. Som exempel på en resurs kan vara ett dokument eller en bild, en tillfällig tjänst, en samling av resurser eller ett icke-virtuellt föremål (t.ex. en människa). Kort sagt så är en resurs ett dataobjekt som kan ges ett namn och som behövs kommas åt av en klient. En viktig egenskap av REST är förmågan att identifiera och lokalisera varje resurs.

Ur en formell synpunkt så kan man beskriva en resurs R som en tillfälligt varierande medlemsskapsfunktion $M_R(t)$ där tiden t är en avbildning till en mängd av ekvivalenta entiteter eller värden. Värdena i mängden kan vara representationer av resurser (eng. resource representations) och/eller resursidentifierare (eng. resource identifiers). En resurs kan också avbildas på en tom mängd. Det tillåter att man kan göra hänvisningar till en resurs före den har skapats. Vissa resurser kan vara statiska, när man vid någon tidpunkt undersöker dom så avbildar de alltid samma värdemängd. Andra mängder kan variera mycket i deras värden under ett visst tidsförlopp. Det enda som krävs av en resurs är att

innebörden bakom avbildningen är statisk. Orsaken till detta är att innebörden är det enda som skiljer en resurs från en annan.

I interaktioner mellan komponenter så använder sig REST av en resursidentifierare (t.ex. URI) för att känna igen resurser. För att komma åt och manipulera en resurs så används det allmänna gränssnittet av en sammankopplare. Man kallar den komponent som tilldelade resursidentifieraren och som gjorde det möjligt att komma åt resursen för den namngivande auktoriteten. Den har bördan att upprätthålla giltigheten av avbildningen så att medlemsskapsfunktionen inte ändrar.

Traditionella hypertextsystem använder sig av unika noder eller dokumentidentifierare som ändras varje gång informationen ändras. Istället använder de sig av länkservrar för att upprätthåller referenser, som är åtskiljda från innehållet. En centraliserad länkserver är mycket dålig idé när man tar i beaktan den stora skalan och multi-organisationella egenskapen av webben. I REST så väljer den som skapar resursen en resursidentifierare som bäst övensstämmer med det koncept som skall identifieras.

4.2.1.2 Representationer

En resurs kan vara nästan vilken sorts av information som helst och en representation används till att ge den en strukturerad form eller typ. Representationen består av en mängd bytes som följer en standard om hur de skall struktureras. På detta vis så kan de riktiga värdena sparas i en databas, vara resultatet av en beräkning eller text som har lästs från en fil, men representationen av denna data kan se helt annorlunda ut. I en instans så kan representationen av t.ex. finansdata vara en tabell i XML, i en annan representation ett cirkeldiagram i ett PDF eller Excel dokument, men den ursprungliga datan skulla förbli oförändrad. En komponent utför handlingar på en resurs genom att använda sig av en representation, som tar vara på det nuvarande eller kommande tillståndet av resursen, som den överför mellan komponenterna.

En representation består av data och metadata. Metadata är data som noggrannare beskriver datan, till exempel datatypen. Ibland så består representationen av metadata som beskriver metadatan. Syftet är att man skall använda sig av denna metadata av metadata för att göra det möjligt att verifiera meddelandet. Metadatan grupperas i formen av ett namn och värdepar. Namnet skall överensstämma med en standard som beskriver strukturen och semantiken av värdet. Svarmeddelanden kan även bestå av både representations- och resursmetadata.

Kontrolldatan har som uppgift att beskriva avsikten med meddelandet mellan komponenterna, d.v.s. orsaken bakom förfrågningen eller innebörden av ett svar. Kontrolldatan används också till att parametrisera förfrågningar och förbigå det normala beteende av sammankopplande element. Till exempel, beteendet av cachen kan ändras av kontrolldatan i ett förfrågnings- eller svarsmeddelande.

Beroende på meddelandets kontrolldata så kan en given representation indikera det nuvarande tillståndet av den förfrågade resursen, det väntade tillståndet för den förfrågade resursen, värdet av någon annan resurs eller en representation av något feltillstånd av ett svar. Som exempel så skapandet på distans av en resurs kräver att skaparen skickar en representation till servern. På det sättet så skapas det ett värde för den resursen, som senare kan nås via en förfrågning. Om en resurs består av flera representationer så kan en innehållsförhandling (eng. content negotiation) ske för att välja den bästa representationen som skall inkluderas i meddelandet.

Dataformatet av en representation kallas mediatyp. I ett meddelade kan man inkludera en representation som mottagaren behandlar på basen av meddelandets kontrolldata och mediatyp. Sammansatta mediatyper kan användas för att skicka flera representationer i samma meddelande.

Designen av mediatypen kan direkt påverka hur användaren ser på prestationen av ett distribuerat hypermediasystem. All data som måste tas emot före mottagaren kan börja rendera representationen ökar interaktionens svarstid. Ett dataformat som lägger den viktigaste informationen först, så att man kan börja rendera datan medan man väntar på mera, resulterar i en mycket bättre prestationen sett ur användarens synvinkel. Till skillnad i ett system där man först

måste vänta på att all data har tagits emot före man kan börja renderingen.

Som exempel kan ges en webbläsare som stegvis renderar ett stort HTML dokument, samtidigt som dokumentet laddas, sett ur användarens vinkel ha en märkbart bättre prestation. Motsatsen till en webbläsare som väntar på att hela dokumentet laddas före sidan renderas. Den första ser ut att ha, ur användarens synvinkel, en bättre prestation även om nätverkets prestationen är den samma i båda fallen.

4.2.2 Sammankopplare

En sammankopplare (eng. connector) är en abstrakt mekanism för som sköter om kommunikation, koordinering och kooperation mellan komponenter. REST använder sig av olika typer av sammankopplare för att enkapsulera aktiviteter. Aktiviteter som uppgiften att accessera resurser och överföring av resursrepresentationer. Sammankopplarna representerar ett abstrakt gränssnitt för komponenternas kommunikation. De gör uppgiften enklare genom att gömma kommunikationsmekanismerna och de underliggande implementationen av resurserna. Om användarna endast har tillgång till systemet via det abstrakta gränssnittet så kan implementationen ändras utan att det inverkar på användarna. En sammankopplare sköter om nätverkskommunikationen för en komponent.

Sammankopplare	Moderna exempel på Webben
Klient	libwww, libwww-perl
Server	libwww, Apache API
Cache	Webbläsarens cache, Akamai cache nätverk
Avgörare	Bind
Tunnel	SOCKS, SSL efter HTTP CONNECT

Lista 2: REST sammankopplare

Alla interaktioner som sker i ett REST system är tillståndslösa. Det betyder att varje förfrågning måste innehålla all den information som är nödvändig för att

förstå förfrågningen. Den skall vara helt och hållet självständig i förhållande till de föregående förfrågningarna. Detta ger i sin tur ger upphov till fyra egenskaper: 1) den tar bort behovet att sammankopplarna måste upprätthålla applikationernas tillstånd mellan olika förfrågningar, 2) den tillåter att interaktionerna kan processeras parallellt utan att tvinga den processerande mekanismen att förstå innebörden av en interaktion, 3) den tillåter att en förmedlare kan se på och förstå en isolerad förfrågning, 4) den tvingar också att all information som påverkar återanvändbarheten av ett svar, som finns lagrat i cachen, att existera i varje förfrågning.

Sammankopplargränssnittet påminner mycket om en procedural anropning, men har många viktiga skillnader när det gäller hur man passerar argument och resultat. Inparametrarna består av förfrågningens kontrolldata, en resursidentifierare som anger målet av förfrågningen och en valbar representation. Utparametrarna består av svarets kontrolldata, valbar resursmetadata och en valbar representation. Ur en abstrakt synvinkel så är anropningen synkron, men både ut- och inparametrarna kan passeras som dataströmmar. Detta leder till att man kan börja processera före parametrarnas värde är helt kända, på det sättet kan man undvika fördröjningen i behandlingen av stora dataöverföringar.

De huvudsakliga typerna av sammankopplare är klienter och serverar. Skillnaden mellan de två olika typerna är att klienten påbörjar kommunikationen genom att göra en förfrågning medan servern lyssnar efter inkommande förfrågningar. Servern svarar sedan på klientens förfrågningar och på det viset ger tillgång till tjänsterna som servern erbjuder. En komponent kan innehålla både klient- och serversammankopplare.

Den tredje typen av sammakopplare, cachesammakopplaren, finns vid gränssnittet av en klient eller en serversammankopplare. Funktionen som den skall uppfylla är att lagra svar i en cache, om det är möjligt att göra, så att svaret kan sedan återanvändas i ett senare skede. Cachen utnyttjas av en klient för att undvika att upprepa en förfrågning eller av en server för att undvika att upprepa skapandet av ett svar. Syftet är att minska på interaktionernas svarstider. En cache är oftast implementerad inom adressområdet av sammankopplaren som utnyttjar den.

Vissa cachesammankopplare är delade med andra. Det betyder att ett lagrat svar kan användas som svar till andra klienter, än till den som svaret ursprungligen var ämnat åt. Genom att dela cachen så kan man på ett effektiv sätt minska effekten av en mycket snabb ökning i belastningen av en server. Det negativa är att det kan också leda till problem om det lagrade svaret inte överensstämmer med det som skulle ha erhållits av en ny förfrågning.

En cache kan bestämma graden av hur bra det går att lagra ett svar, på grund av att gränssnittet är allmänt istället för att vara specifikt för varje resurs. Normalt är svaret till en återfående förfrågning (eng. retrieval request) lagringsbar och svaret till andra förfrågningar inte lagringsbara. Om någon sorts av användarautentisering är en del av förfrågningen, eller om svaret indikerar att den inte skall delas med andra, då är svaret endast lagringsbart av en icke-delbar cache. En komponent kan kringgå de normala inställningarna genom att inkludera kontrolldata. Kontrolldatan kan specificera om interaktionen är lagringsbar, inte lagringsbar eller endast lagringsbar för en begränsad tidsperiod.

En avgörare (eng. resolver) översätter ofullständiga eller fullständiga resursidentifierare till information om nätverksadressen som behövs för att etablera en uppkoppling mellan komponenter. Till exempel de flesta URI:na inkluderar ett DNS värdnamn som ett sätt att identifiera resursens namngivningsauktoritet. För att starta en förfrågning så kommer en webbläsare att ta ut värdnamnet från URI:n och använda sig av en DNS avgörare för att få en IP-adress för den auktoriteten. Ett annat exempel är att vissa sorter av identifikationsscheman kräver att en förmedlare översätter en permanent identifierare till en temporär adress för att komma åt den identifierade resursen. Användning av en eller flera förmedlare kan förlänga livet, genom förmedling, av en referens till en resurs men det leder till en ökning i svarstiden för en förfrågning.

Den sista typen av sammankopplare är en tunnel. En tunnel är en sammankopplare som endast vidareförmedlar kommunikation över en uppkopplingsgräns, som en brandmur eller en nätverksbrygga. En viktig orsak till att den är en del av REST, istället för att vara abstrakt som en del av nätverksstrukturen, är det att vissa

REST komponenter kan dynamiskt byta beteende mellan att vara en aktiv komponent till det att vara en tunnel. Som ett exempel när en HTTP proxy byter beteende till en tunnel, som svar på en CONNECT förfrågning. På det sättet så tillåts det direkt kommunikation med en avlägsen server genom att använda sig av ett annat protokoll, som TLS, som inte tillåter användning av proxyn. Tunneln försvinner när båda ändorna slutar att kommunicera med varandra.

4.2.3 Komponenter

En komponent kan beskrivas som en abstrakt enhet som består av mjukvaruinstruktioner och ett internt tillstånd och som har ett gränssnitt som åstadkommer en transformation av data. REST komponenter kan ordnas enligt vad deras funktion är, se Lista 3.

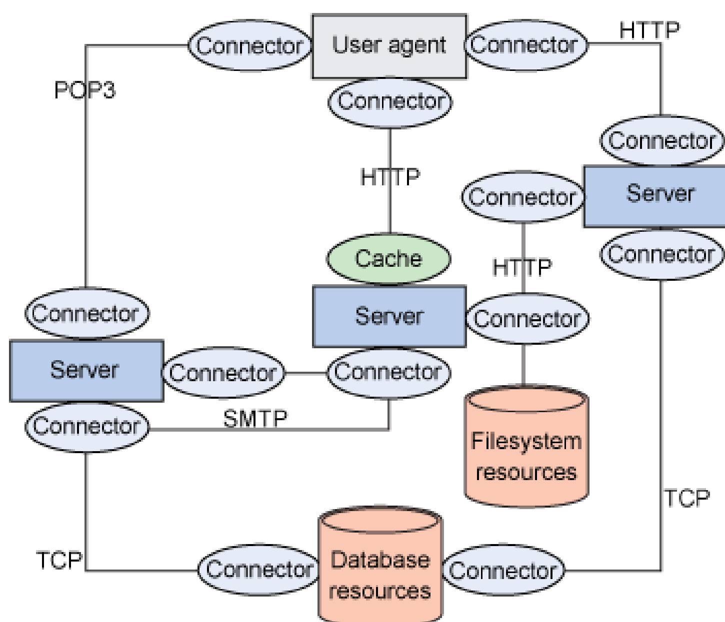
Komponenter	Moderna exempel på Webben
Ursprungsserver	Apache httpd, Microsoft IIS
Gateway	Squid, CGI, Reverse proxy
Proxy	CERN Proxy, Netscape Proxy, Gauntlet
Användaragent	Mozilla Firefox, Google Chrome, Internet Explorer, Opera

Lista 3: REST komponenter

En användaragent (eng. user-agent) utnyttjar en klientsammankopplare för att göra en förfrågning och fungerar sedan som mottagaren av förfrågningen. Det vanligaste exemplet av en användaragent är en webbläsare.

En ursprungsserver (eng. origin server) använder sig av en serversammankopplare för att sköta om namnutrymmet av en förfrågad resurs. Det är den enda riktiga källan för sina egna resursers representationer och måste vara den slutgiltiga mottagaren av alla förfrågningar som har målet att ändra på värdet av källans resurser. Varje ursprungsserver erbjuder ett allmänt gränssnitt åt sina tjänster, i formen av en resurshierarki. Detaljerna hur resurserna har implementerats är gömda bakom gränssnittet.

Förmedlande komponenter fungerar både som en klient och som en server för att kunna vidarebefordra, vid behov utföra en översättning av, förfrågningar och svar. En proxy komponent är en förmedlare som har valts av en klient för att fungera som ett gränssnitt vid enkapsulering av andra tjänster, översättning av data, förbättring av prestanda eller för att förbättra säkerheten. En gateway (även kallad en reverse-proxy) komponent är en förmedlare som tvingas av nätverket eller av ursprungsservern att förse ett gränssnitt för enkapsulering av andra tjänster, översättning av data, förbättring av prestanda eller för att förbättra säkerheten. Skillnaden mellan en proxy och en gateway är att klienten bestämmer när den använder sig av en proxy.



Figur 3: : Interaktion mellan REST entiteter

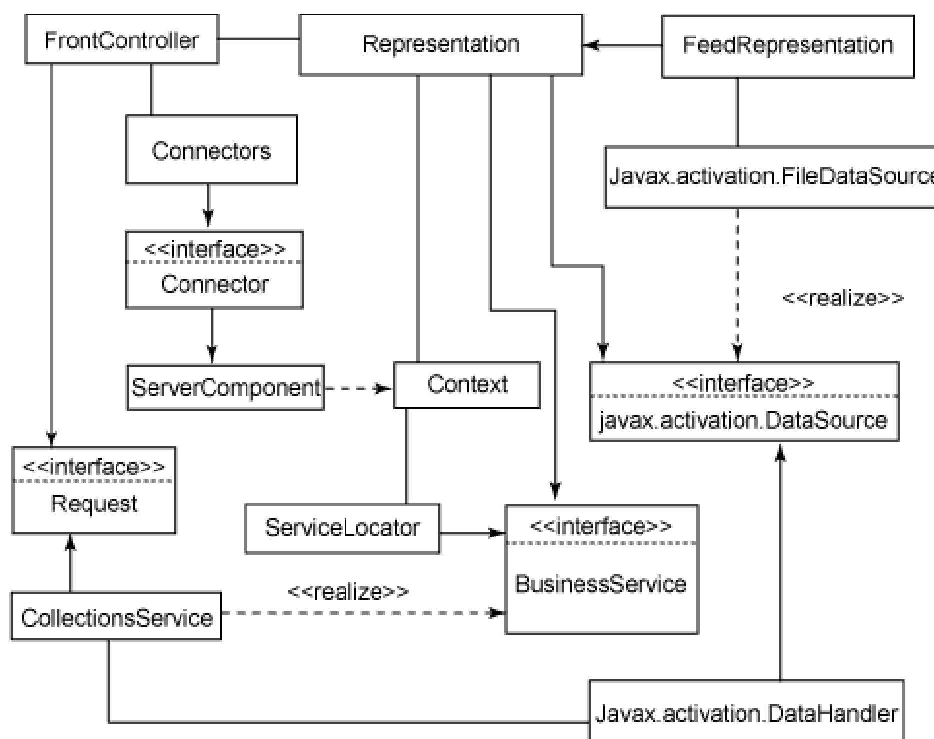
I figuren, se Figur 3, visas det hur olika entiteter (sammankopplare, komponenter och resurser) samarbetar med varandra i ett nätverk. Förfrågningarna skickas och tas emot av sammankopplare som är bunda till en entitet, som t.ex. en server eller en databas. Flera sammankopplare kan sköta om en entitets kommunikation, där varje sammankopplare tar hand om ett givet protokoll.

4.3 Användning

En implementation som följer de principer som REST har lagt fram brukar kallas för en RESTful implementation. Det finns flera olika implementationer som följer REST och som några exempel kan ges: Twitter API, tjänster från Yahoo!, Flickr, Atom Publishing Protocol (AAP).

Atom Publishing Protocol är ett HTTP-baserat protokoll vars huvudsakliga uppgift är att göra det enkelt att skapa nya resurser, editera resurserna, ta bort resurser och hämta resurser som finns inne i en samling (eng. collection). Protokollet använder sig av de operationer som HTTP erbjuder (GET, PUT, POST och DELETE) för att överföra dokument och samlingar dokument som innehåller resurser.

Som ett exempel på en enkel implementering, se Figur 3, av ett applikationsramverk som utnyttjar AAP.



Figur 2: Komponenter av ett REST/APP applikationsramverk

I detta applikationsramverk så fungerar FrontControllern som en huvudklass. Den

tar emot förfrågningar och interagerar med sammankopplare för att behandla ServerComponent objekt. ServerComponent objekten används för att hitta BusinessService objekt som sköter om företagslogiken av varje förfrågning. Resurser är definierade av Representation gränssnittet.

5. Sammanfattning

Hur skulle det gå att använda sig av REST i ett gridsystem? Vad skulle vara fördelarna och nackdelarna?

Varför inte använda sig av några andra teknologier som t.ex. WSRF?

WSRF vs REST.

Referenser

1. Bieberstein, N.; Fiammante, M.; Jones, K.; Shah R. : Service-oriented architecture compass: business value, planning, and enterprise roadmap, IBM Press, 2006.
2. Brown, P. F.; Laskey, K.; MacKenzie, C. M.; McCabe, F. : Reference Model for Service Oriented Architecture 1.0 : OASIS Standard, 12 October 2006, http://opengroup.org/projects/soa/doc.tpl?gdid=10632
3. Ekblom, Richard : Applied Representational State Transfer, Umeå University, 2011
4. Fielding, Roy Thomas: Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Hämtad 9.2.2011

5. Fielding, Roy T.; Taylor, Richard N. : Principled Design of the Modern Web Architecture. University of California, Irvine, 2002, http://www.ics.uci.edu/~taylor/documents/2002-REST-TOIT.pdf
6. Foster, I.; Kesselman : The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 2004. ISBN-10: 1-55860-933-4
7. Friberg, D.; Gyllander, K.; Mondélus, M. : SOA – Tänk efter före: Ett organisationsförberedande ramverk inför implementeringen av en tjänsteorienterad arkitektur. Examensarbete, Göteborgs Universitet och Chalmers Tekniska Högskola, 2007, http://gupea.ub.gu.se/bitstream/2077/4623/1/SSYSM_IA7400_V07_%20Friberg%2cDaniel_Gyllander%2cKristina_Mondelus%2cMond%c3%a9sir_SOA%20_%20T%c3%a4nk%20efter%20f%c3%b6re.pdf
8. Hanson, J. : Writing REST services. DeveloperWorks, 2007, http://www.ibm.com/developerworks/xml/tutorials/x-restatompp/x-restatompp-pdf.pdf
9. Kuba, Martin; Krajícek, Ondrej : Literature search on SOA, Web Services, OGSA and WSRF, 2007, http://www.ics.muni.cz/~makub/soap/reserse_wsrf.pdf
10. Pautasso, C.; Zimmerman, O.; Leymann, F.: RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision, 17th International World Wide Web Conference, 2008, http://www.jopera.org/files/www2008-restws-pautasso-zimmermann-leymann.pdf
11. Mahmoud, Qusay H. : Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI). Oracle Technology Network Article, 2005, http://www.oracle.com/technetwork/articles/javase/soa-142870.html Hämtad 4.4.2011