LEIF JOHANSSON

Sektionen för IT och media

TEKNISK SAMORDNING

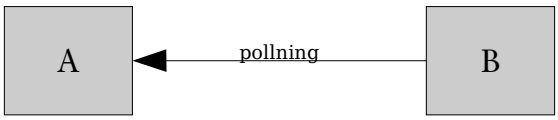
2005 05 11

JEvent: Teknisk dokumentation för Stockholms universitet

Inledning

JEvent (Jabber Events) är ett system för meddelande-baserad infrastructur (MOM – message oriented middleware) baserat på XMPP (RFC xxxx). JEvent består både av klientbibliotek (api) och flera tjänster. Syftet med JEvent är att göra det enkelt att hantera kommunikation mellan processer eller applikationer som är asynkrona till sin natur. Exempel är alla situationer när någon process behöver notifiera en annan process om en händelse:

Denna situation är mycket vanlig. Att skicka ett meddelande mellan system kräver någon form av infrastruktur. Om denna infrastruktur saknas finns det bara en lösning, nämligen att polla B från A vilket är ineffektivt.



Genom att bygga en infrastruktur för asynkrona meddelanden uppnår man flera positiva effekter:

- Snabbare respons: A kan omedelbart informera B om att något har hänt.
- Processen kan övervakas: infrastrukturen i sig består av tekniska system som kan övervakas.
- Meddelanden kan loggas i systemet vilket också ger möjlighet till 'playback' av tidigare händelser.

De system som skickar eller tar emot meddelanden kallas *agenter*. En agent kan antingen vara *producent* (skicka meddelanden) eller *konsument* (ta emot) eller både och. Detta kanske verkar överdrivet trivialt men det är det inte. Att både konsumera och producera meddelanden ställer krav på antingen intern event-

Postadress:
Teknisk Samordning
Sektionen för IT och Media
Stockholms universitet

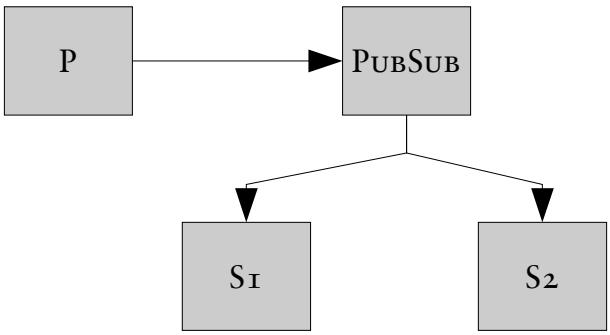
Leif Johansson
leifj@it.su.se

hantering i agenten eller någon annan form av parallellism (tex trådar) vilket i sig ökar komplexiteten i en agent. Det enda undantaget är om agenten endast producerar ett nytt meddelande i samband med att något (antingen ett meddelande eller kommunikation via någon annan kanal) kommer in till agenten.

Det är alltså relativt enkelt att bygga pipelines av agenter som skickar meddelanden till varandra. Detta mönster är viktigt eftersom det gör det enkelt att isolera olika delsystem från fördröjningar som kan uppkomma. Många underliggande processer (tex kontohantering, AFS-volymhantering etc) är till sin natur asynkrona. Det kan vara omöjligt att förutsäga om det kommer att gå snabbt att genomföra en operation eller om omständigheterna gör att det ibland tar längre tid.

Genom att enkapsulera sådana system och bygga ett asynkront gränssnitt mot dom så kan man göra felhanteringen mycket enklare.

Meddelanden behöver inte bara vara punkt-till-punkt. Många applikationer kräver att fler än en agent får samma meddelande av viss typ. Denna typ av meddelande-passering (många-till-en) liknar ofta den klassiska usenet-modellen där servern har flera news-grupper. Ett meddelande skickas från en postare till en newsgrupp där en eller flera prenumeranter får information om att ett nytt meddelande finns att hämta. Denna modell kallas ofta *publish-subscribe* eller *PubSub*



Teknisk implementation på SU: JEvents

JEvents är både namnet på infrastrukturen för asynkrona meddelanden (tjänsten) och namnet på det perl-paket som utgör ramverket för att skriva JEvents-agenter (i perl). I framtiden kommer andra api-bibliotek att utvecklas.

Alla meddelanden mellan agenter skickas via ett meddelandepasseringssystem som är baserat på en xmpp-server som implementerar JEP-0060, xmpp-versionen av PubSub: http://www.jabber.org/jeps/jep-0060.html. Det finns flera fördelar med att använda xmpp som bas för meddelandesystem:

- Vanliga jabber-klienter kan användas som människa-interface till systemet. En agent kan ta emot och behandla meddelanden som kommer in via olika kanaler (på olika sätt). Tex kan en agent fungera i en group-chat och ta emot kommandon från flera användare.
- Säkerhetsinfrastruktur: SASL, TLS och S/MIME för säkerhet på meddelandenivå.
- Inbyggd store-and-forward via delayed delivery – JEP-0091 – (http://www.jabber.org/jeps/jep-0091.html). Detta betyder att agenter kan vara offline under kortare tid utan att meddelanden tappas bort sålänge jabberservern är uppe.

Inter-domän-trafik. Precis som en vanlig jabber-tjänst är det möjligt att koppla ihop eventsystem från olika organisationer.

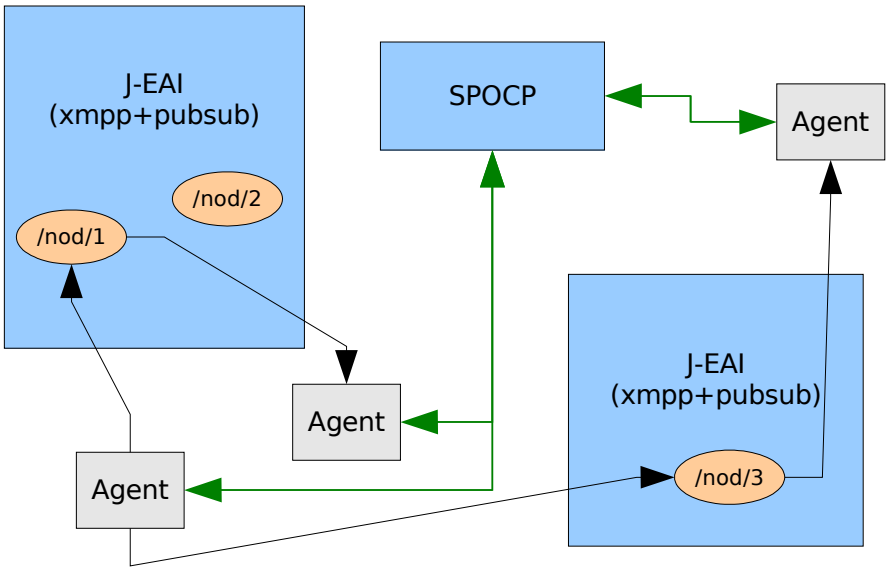
JEvents kräver en relativt komplett implementation av JEP-0060 (PubSub). Vi använder ejabberd med pubsub-modulen från J-EAI-projektet.

Det är klart möjligt att köra flera separata server-instanser inom samma administrativa domän. Vissa agenter kan ställa höga genomströmingskrav på servern och kan därmed ställa krav på fler servrar. Genom inter-domän-trafik (s2s) så fungerar meddelanden mellan server-instanser utan problem.

Perl-modulen JEvent implementerar en specifikation (se nedan) av hur agenter ska fungera. Det är möjligt (och antagligen önskvärt) att på sikt ta fram andra implementationer av samma specifikation.

JEvents består av ytterligare en komponent: SPOCP. Det finns flera operationer som kräver någon form av behörighetskontroll:

- när en agent får en presence-subscribtion-requests
- när en agent inbjuds till en groupchat
- när en agent får ett kommando från en människa



Denna bild illustrerar hela systemet – multipla server-instanser, agenter och noder samverkar kring en policy-motor till att bilda ett meddleandesystem:

Användningsfall

I detta avsnitt går vi igenom några (realistiska) användningsfall av JEvent som visar på vad man kan göra med teknologin.

Syslog-bot

Agenten kör som log-mottagare på en syslog-ng-server. Nya loggrader kommer in till agenten på STDIN. Agenten upprätthåller ett antal aktiva *sessioner*. Varje session är associerad med antingen en chat-session eller med en groupchat/multi-user chat. Varje session har någon form av filter-uttryck som avgör vilka loggrader som ska skickas vidare till sessionen.

Varje gång en rad kommer in matchas den mot de aktiva sessionerna och skickas vidare. Agenten svarar också på kommandon som tillåter att användare ändrar på det aktiva filtret.

Sessioner öppnas genom att chat-sessioner eller groupchat/MUC-sessioner öppnas mot agenten på vanligt sätt.

LogDB med PubSub

Även denna agent kör som log-mottagare på en syslog-server men denna agent skickar log-entryn vidare som XML till en pubsub-nod. Genom att köra flera instanser av agenten mot olika pubsub-noder kan log-meddelanden från olika källor skickas till olika noder.

Genom att subscriba på dessa noder kan andra agenter få information om log-evetns. En av dessa agenter sparar events till en databas för senare analys (logdb).

Allokering av UID/GID

Idag allokeras numeriska UID:er och GID:er manuellt eller genom AFS-PTS. Detta medför problem och en risk för kollisioner och dubbelallokering. En lösning är att implementera en allokerings-tjänst som en *singleton*. En singleton är ett objekt som garanterat bara finns i ett exemplar. Agenten är i övrigt väldigt enkelt. Den svarar helt enkelt på (behöriga) kommandon som begär allokering av nästa uid/gid.

Jabber-VOS

Denna agent skapar och provisionerar (bla sätter quota för) volymer och monterar dem på angett ställe i AFS-filträdet. En sådan agent ersätter i princip vos och gör att man får bättre och mer finkornig behörighetskontroll på volym-operationer. I den enklaste formen svarar agenten bara på kommandon men man kan även tänka sig att agenten lyssnar på PubSub-events som informerar om att användare aktiverats i SUKAT. Som ett led av aktiveringen ska användaren få en volym vilket alltså denna agent skulle hantera.

HostDB-agent

Varje gång en zon uppdateras i CVS eller i HostDB måste man begära omstart av denna zon för att ändringar ska ta effekt i DNS-servrar och DHCP-servrar. Detta sker idag genom ett webbgui eller genom att man loggar in på car.su.se och kör ett kommando. Genom att placera applikationslogiken för omstart av en zon i en agent som både svarar på kommandon och på pubsub-events kan dels administratörer starta-om zoner utan att behöva gå genom ett webbgui och dels kan omstart av zoner ske som följd av att andra händelser sker.

Apachectl

Varje gång en vhost skapas måste motsvarande apache-instans startas-om. Detta är en felkännslig process som desstuom bara idag får göras av ett fåtal tekniker. Genom att placera applikationslogiken för korrekt omstart i en agent minskas felen vid omstart och desstom kan fler få rätt att starta-om (beroende på ägare till vhost) apache-servrar.

AFS-FAM (vos-release-tjänst)

AFS stödjer inte multimaster-volymer – en fil ligger antingen på en RW-volym och förändringar slår igenom omedelbart, eller så ligger den på en RO-volym som kräver en manuell vos release för att replikeras till alla RO-replikor. Eftersom vos release är en priviligierad operation (bara användare i UserList på filservrarna får

göra vos release) är RO-replikor bara användbara för data som ändras sällan och dessutom av administratörer med utökade privilegier.

AFS-FAM (FAM - file alteration monitor är ett program som löser motsvarande problem för lokala filsystem) består av tre agenter. En agent publicerar events när något har ändrats i en katalog – detta triggas antingen av ett kommando eller på annat sätt. Den andra agenten lyssnar på dessa events och omvandlar dem till events på volymnivå. Den tredje agenten lyssnar på volym-nivå-events som signalerar förändring av någon data i en volym och gör vos release. Denna agent ansvarar för att göra rate-limiting och annan policy.

Den andra agenten blir dessutom i princip en självlärande mountpoint-databas.

JEvent agent-specifikationen (v 1.0)

En JEvent-agent är en XMPP-klient som kommunicerar med andra agenter med PubSub-meddelanden över xmpp och med andra xmpp-klienter (som inkluderar människor och andra system) via vanliga xmpp-meddelanden.

Meddelandeflöden

Följande tabell visar hur agenten ska hantera olika meddelanden: Meddelanden som inte är listade kan hanteras av agenten men framtida versioner av denna specifikation kan komma att inkludera dessa. Om man vill undvika framtida konflikter ska man bara implementera dessa medelandeflöden i agenten:

Meddelande/ Namespace	*Specifikation*
Pubsub#event	Notifiering om ett event (tex att något publicerats) på en pubsub-nod. Agenten kan välja att hantera eller ignorera ett event. Innehållet i ett event är alltid XML och varierar från applikation till applikation. Ett event skickas som xml-payload till ett <message/> meddleande men ska alltså inte behandlas på samma sätt som andra vanliga <message/>-tags.
<presence/>	Ett presence-meddelande som är av typen 'subscribe' är en begäran från en annan xmpp-klient att subscriba på agentens presence – mao att lägga till agenten till någon roster. Agenten ska göra berhörighetskontroll av denna begäran och skicka ett presence-meddelande av typen 'subscribed' eller 'unsubscribed' beroende på om spocp-servern tillät operationen eller ej.
jabber:x:conference	Ett vanligt meddelande som innehåller en sub-tag med detta namespace är en inbjudan till en groupchat eller en multi-user chat. En agent ska göra en kontroll av att avsändaren har rätt att inbjuda agenten till rummet via spocp. Om inte så ska ett fel skickas i ett vanligt meddelande (i text-bodyn) av typen 'normal', annars ska agent en ansluta till konferensrummet och skicka ett hälsningsmeddelande.

Meddelande/ Namespace	*Specifikation*
<message/>	Detta är ett textmeddelande (kanske från en jabber-klient). Om meddelandet är av typen 'chat' eller 'normal' så bör texten parsas innehållet enligt regexp:en `'^\s*(\S+)\s*(.*)\s*$'`. Den första komponenten ska tolkas som namnet på en funktion eller kommando. Övriga delar ska tolkas som argument till funktionen. Om namnet på funktionen inte motsvarar någon för agenten känd funktion ska agenten svara med ett meddelande *av samma typ* med ett textuellt felmeddleande. Om kommandot finns så ska agenten göra ett behörighetsanrop till en spocp-server och beroende på resultatet utföra kommandot eller svara med ett meddelande som innehåller ett fel-meddelande. Om typen är 'groupchat' så sker samma processning med skillnaden att kommandosträngen måste prefixas av '<alias>:' där <alias> är agentens alias i konferens-rummet. Om kommandot utförs ska resultatet (det är upp till agenten att avgöra vad som utgör resultetet och hur detta ska formatteras) skickas som innehållet i ett text-meddelande av samma typ. Ett meddelande ska alltid besvaras utom i fallet då meddelandet är ett groupchat-meddelande som inte uppfyller ':'-addressering till agenten. Sådana meddelanen ska ignoreras tyst.

Exempel på kommandoprocessning

Exempel 1:

```
<message from='foo@su.se/Psi'
         to='agent@jevent.it.su.se/JEvent'
         type='chat'>
   <body>ls /tmp</body>
</message>
```

Detta meddelande ska tolkas som att kommandot 'ls' anropas med argumentet '/tmp'. Evt svar eller felmeddleanden skickas tillbaka till 'foo@su.se' som ett meddelande av typ 'chat'.

Exempel 2:

```
<message from='rummet@conference.su.se/foo'
         to='agent@jevent.it.su.se/JEvent'
         type='groupchat'>
  <body>agent: ls /tmp</body>
</message>
```

Detta ska tolkas som att foo (nick för foo@su.se) skickar kommandot 'ls /tmp' till agent. Evt svar eller felmeddleanden skickas till 'rummet@conference.su.se' som ett meddelande av typen 'groupchat'.

Exempel 3:

```
<message from='rummet@conference.su.se/foo'
         to='agent@jevent.it.su.se/JEvent'
         type='groupchat'>
  <body>hej hopp</body>
</message>
```

Detta meddelande ska agenten ignorera eftersom det inte är direkt addresserat (genom ':'-notationen) till agenten.

Behörighetsanrop

Det finns ett flertal fall då en agent ska göra ett anrop till en spocp-server för att kolla behörighet. Dessa anrop ska använda en av två regelsyntaxer; en för olika sorters presence-hantering (inklusive begäran om att agenten ska ansluta till ett konferensrum) och en syntax som används när agenten ska utföra ett kommando.

Regelsyntax för xmpp-anrop

Denna syntax används vid kontroll av presence-subscription samt invitering till konferensrum. Den används

```
(jevent (method)
        (from)
        (to))
```

Regelsyntax för kommandon

När ett kommando ska utföras ska en agent göra ett spocp-anrop med en query som uppfyller följande schema:

```
(jevent (command)
        (from)
        (to))
```

Exempel

Om klienten 'foo@su.se' har skickat kommandot 'ls' till 'agent@jevent.it.su.se' så blir spocp-frågan:

```
(jevent (command ls)
        (from foo@su.se)
        (to agent@jevent.it.su.se))
```

Om foo@su.se istället försöker subscriba på presence för agent@jevent.it.su.se blir spocp-frågan:

```
(jevent (method subscribe)
        (from foo@su.se)
        (to agent@jevent.it.su.se))
```

Perl-agenter med JEvent.pm

Detta avsnitt sammanfattar och exemplifierar genom perl-api:et JEvent.pm. Detta api tillåter större frihet än vad JEvent-specifikationen medger men dessa features kommer vi inte att gå igenom här. För mer detaljer bör man läsa JEvent.pod.

En agent skriven med JEvent.pm består i princip av två funktionsanrop:

```
use JEvent;

my $agent = JEvent->new(...);
...
$agent->Run(...);
```

Först skapas en JEvent-instans där parametrar som styr hur agenten kommer att agera initialiseras. I detta steg sker nästan all provisionering av agenten. Anropet till Run-metoden startar agentens request-loop och avslutar normalt inte.

Argumenten till new-methjoden är på formen Type => $value. I många fall ska argumentet till 'Type' vara en CODE-referens; mao ett funktionsargument. Dessa funktioner anropas på olika ställen i agentens request-loop.

Agentens statiska konfiguration (tex den Jabber-ID som ska användas) kan anges via ett Config::IniFiles-object (en INI-fil) som anges via Config-argumentet. Dessa argument har ibland ett annat namn i INI-filen än om de anges som argument till new.

Argument	*Innehåll/Beskrivning*
Commands	Värdet ska vara en HASH-ref där nyckelvärdena är namnet på de kommandon som ska definieras i agenten och värdena är CODE-ref som implementerar resp. kommando. Dessa funktioner får argumenten ($self,$from,$type,$cmd,@args) som är (i ordning): • JEvent-instansen • JID (sträng) som kommandot kommer ifrån • Meddelande-typen som kommandot kom i (ex groupchat) • Namnet på kommandot • Argument till kommandot
ProcessCB	Funktion som anropas (med JEvent-objektet som argument) varje gång agentens selectloop inte gör något annat. Denna funktion får inte göra något som tar lång tid. Exempel på saker som passar här är tex: • läsa data från stdin eller någon annan kanal • bevaka någon extern resurs – tex en fil

Argument	*Innehåll/Beskrivning*

EventCB	Funktion som anropas varje gång en pubsub event kommer in till agenten. En event är en <message/> med en inbäddad <x/> som innehåller datat som publicerats (om noden är konfigurerad att leverera payload med notifieringar – se JEP-0060).
	Funktionen anropas med argumenten ($self,$sid,$msg) som är (i ordning):
	• JEvent-instansen
	• XMPP Sessions-id för meddelandet
	• Meddelandet som innehåller notifieringen
	För att komma åt listan av <item/> som finns i en notifiering använder man följande kod:
	`foreach my $item ($msg->GetItems()->GetItems()) {`
	`...`
	`}`
	Det dubbla anropet till GetItems beror på att $msg har ett barn-objekt (i XML) som heter <items/>. Detta objekt har en lista med <item/>-taggar.

Timeout	Timeout för (bla synkrona) operationer.
JID	Agentens Jabber-ID
Password	Agentens Lösenord – idag stöds bara TLS+plain som auth
DebugLevel	0-3 beroende på hur mycket extra debggning man vill ha
DebugFile	Ange 'stderr' för att få debuggning på STDERR
Host	Default-namn på tjänst (oftast namnet på pubsub-tjänsten)
Node	Default-noden där agenten publicerar data via $self->Publish.
Description	Kort beskrivning av agenten
SPOCPServer	host:port
UseTLS	Använd START_TLS
SSLVerify	Se IO::Socket::SSL
CAFile	Se IO::Socket::SSL
CADir	Se IO::Socket::SSL
ProcessTimeout	Hur ofta (i sekunder) som ProcessCB körs

Ett exempel som visar hur detta hänger ihop är echoagent i JEvent-distributionen. Perl-implementationen av jevents (JEvents.pm) medger att vissa konfigurationsvariabler läses in från en INI-fil (tex är det lämpligt att lagra lösenord mm där). Namnet på variablerna i INI-filen skiljer sig delvis från tabellen ovan (obs att case ibland skiljer sig åt av historiska skäl):

JEvent.pm	*INI*
Host	[PubSub]Host
Node	[PubSub]Host
JID	[JEvent]JID

JEvent.pm	*INI*
Password	[JEvent]Password
CAFile	[SSL]cafile
CADir	[SSL]cadir
SPOCPServer	[SPOCP]Server

Interfacet till ini-filen Ett enkelt exempel på en INI-fil för en agent:

```
[SSL]
cafile=/etc/ssl/pcacert.crt


[SPOCP]
Server=spocp.su.se:3456

[JEvent]
JID=test@cdr.su.se/JEvent
Password=secret

[PubSub]
Host=pubsub.cdr.su.se
```

Detta exempel är tänkt för en agent med JID test@cdr.su.se (observera att en resurs alltid bör anges) som per default publicerar events till pubsub.cdr.su.se och som använder spocp.su.se:3456 som SPOCPServer.

Om agenten behöver tillgång till andra konfigurationsparametrar så kan man anropa funktionen `$jevent->cfg($section,$key)`. Första argumentet anger avsnittet i INI-filen och andra parameternyckeln. Varje agent bör allokera en egen sektion och ha all konfiguration där. Detta gör det möjligt att dela INI-filer mellan olika agenter och applikationer.

Interaktiva agenter och Webbinterface

En agent kan ses som hemvist för en avgränsad uppsättning funktioner med flera olika gränssnitt. Ett asynkront gränssnitt där agenten reagerar på events via PubSub och ett synkront gränssnitt där agenten svarar på kommandon. I vissa fall behöver agenten samverka med webb-applikationer – tex när agenten utgör bakända till ett webbgui.

Webbgränssnittet behöver då ett sätt att anropa funktioner i en agent. Detta kan ske genom att generera events eller kommandon till agenten. JEvent (perl-klassen) innehåller funktioner som gör det enkelt att anropa en agent från ett CGI. Istället för att anropa `$jevent->Run()` använder man istället $jevent->Connect() följt av ett eller flera anrop till metoder i JEvent. Som exempel följande kodsnutt som visar hur ett webb-gränssnitt skapar och publicerar ett event skapat från CGI-variabler:

```
#!/usr/bin/env perl

use CGI;
use JEvent;
use Config::IniFiles;
```

```
# Läs konfiguration och skapa agent resp. CGI-objekt
my $ini = Config::IniFiles->new(-file=>'agent.ini');
my $jevent = JEvent->new(Config=>$ini);
my $cgi = CGI->new();

# Skapa förbindelse till XMPP-servern och logga in
$jevent->Connect();

# Skapa och publicera lite XML
my $now = time();
my $p = $q->param('data');
$jevent->Publish(Node=>$q->param('node'),Content=><<EOC);
<foo time='$now'>
  <data>$data</data>
</foo>
EOC

# Logga ut och stäng förbindelsen
$jevent->Disconnect();
```

Den XML som publiceras spelar naturligtvis ingen roll för exemplet. En agent som behöver någon form av strukturerad data från ett webbgränssnitt eller direkt från en annan XMPP-klient kan använda Jabber Data Forms (JEP-0004) som ger ungefär samma funktionalitet som HTML-formulär. Få vanliga XMPP-klienter stödjer JEP-0004 utom för vissa speciella tillämpningar. Emellertid är JEP-0004 implementerat i JEvents (perl-klassen). Detta gör det möjligt att bygga agenter som har interaktiva dialoger med ett webbgränssnitt.

Administration av JID för agenter

Varje agent och varje webbgränssnitt behöver en service-användare i SUKAT med associerad principal. Genom att knyta objekt-klassen 'jabberUser' till en sådan service-anvädare kan en eller flera JID associeras till användaren. Vid inloggning till XMPP-servern används lösenordet som hör till principalen som hör till service-användaren som hör till den aktuella JIDen.

Alla sådana service-användare ska ha namn på formen su-j-* för att garantera unika namn i det globala uid-namespacet på SU. Exempel: en användare för testapp@cdr.su.se ser ut såhär i SUKAT:

```
dn: su-j-testapp,dc=jevent,dc=it,dc=su,dc=se
uid: su-j-testapp
jabberID: testapp@cdr.su.se
objectClass: account
objectClass: jabberUser
objectClass: top
```

Detta exempel förutsätter att användare skapas i domänen jevent.it.su.se vilket inte är absolut nödvändigt men underlättar underhåll. Exakt vilken domän som JID tillhör kommer att variera beroende på hur många instanser av JEvent som kommer att finnas – tex kan vissa applikationer kräva separata event-servrar (tex om man behöver extra hög tillgänglighet för vissa tjänster).

Administration av PubSub-noder för agenter

PubSub-noder skapas och administreras via http://www.it.su.se/pubsub. En PubSub-nod har en ägare som har rätt att göra administrativa operationer på noden. Denna rättighet inkluderar inte att publicera data till eller prenumerera på notifikation från noden, mao har ägaren till noden bara administrativa rättigheter. Alla noder som skapas via webbgränssnittet ovan har samma ägare vilket alltså inte har någon praktisk betydelse. Det är naturligtvis möjligt att skapa noder med andra klienter – tex finns det stöd för en pubsub-agent i JEvent som kan göra pubsub-administration via kommandon.

Det finns två roller som en JID kan ha relativt en nod: publisher och subscriber. En publisher har rätt att publicera data till noden. En subscriber får notifieringar från noden. Det finns dessutom ett antal parametrar som kan ställas in per nod som bla styr om notifieringar ska innehålla data, hur länge och hur mycket data som ska sparas och om subscribers ska få notifieringar när nodens konfiguration ändras. Denna uppdelning i publishers och subscribers innebär en enkel accesskontrolls-modell för PubSub som är *inbyggd i standarden* – det är alltså inte möjligt att använda SPOCP för denna accesskontroll.

En publisher kan samtidigt vara subscriber och kom mer då att få en notifiering för varje publicering som görs från agenten. Det vanliga är att en agent inte både är publisher och subscriber för samma nod.

Semantisk frikoppling

Det finns en grundläggande princip som ska råda när man skapar nya noder och syntax för data som ska publiceras:

Syntax för XML-content och namngivning av noder ska så långt det är möjligt vara semantiskt neutrala.

Med *semantik* menas i detta sammanhang 'betydelse eller innebörd av data'. På ett annat sätt kan principen formuleras såhär: data som publiceras till en nod såväl som namnet på noden ska så långt det är möjligt inte bara medge en tolkning. Denna princip illustreras bäst med ett exempel på det motsatta: ett fall av hård semantisk inlåsning.

Tänk på vos-release-tjänsten som beskrivs ovan. När vi tänker på hur data som den andra agenten – den som subscribar på events om modifieringar av filsystemet och omvandlar dem till events om modifieringar av volymer – ska utformas skulle man kunna välja en syntax för XML-content enligt följande exempel:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<release-volume name=”services”/>
```

Denna syntax innehåller ett underförstått antagande att den agent som subscribar på denna typ av events gör en specifik operation (”release-volume”) på volymen. I detta extremt enkla exempel är den semantiska inlåsningen väldigt enkel att bortse ifrån – vi kan ju som agentutvecklare välja att tolka ”release-volume” som något helt annat – tex: ”logga att någon begärde release-volume” men inlåssningen finns där. Det är enkelt att konstruera andra exempel där inlåsningen är hårdare. Tex skulle vi kunna skicka SOAP-anrop över PubSub. Då skulle det vara mycket svårt att inte tolka XML som ett funktionsanrop via just SOAP.

Ett bättre exempel i just detta fall kanske hade varit:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<volume-modification type=”data” name=”services”/>
```

Anledningen till att semantisk inlåsning är ett problem är detta: Vi vet inte när vi skapar en agent och en uppsättning PubSub-events exakt hur dessa kommer att användas i framtiden. Genom att låta mottagaren av ett event (subscirbern) ha ansvaret för att *tolka* den XML som ett event innehåller frikopplar vi publiceraren från mottagaren på ett semantiskt plan på samma sätt som PubSub gör en tidsmässig frikoppling (asynkrona meddelanden) mellan avsändare och mottagare av ett event.

I vårt exempel betyder detta konkret att den andra agenten producerar events som bara signalerar att en volym har ändrats och att det är data (snarare än metadata om volymen) som ändrats. Det är den 3e agenten som *tolkar* detta event som att det föreligger ett behov av att göra ”vos release”. Man kan dessutom enkelt tänka sig andra agenter som tolkar detta event på andra meningsfulla sätt.

Denna frikoppling i tolkning mellan avsändare och mottagare gör det möjligt att susccessivt utveckla agenter i nya riktningar.

Ytterligare ett sätt att förstå semantisk frikoppling är att tänka på PubSub-events som information om att något har hänt (*denna volym har ändrats*) eller att ett visst tillstånd föreligger (*denna volym har nu 3 monteringspunkter*) snarare än som begäran att något ska ske (*gör vos release*) – mao beskrivande snarare än operationella uttryck för förändring.

Alternativ för semantiskt neutral syntax: RDF och OWL

När man designar semantiskt neutral syntax för pubsub-events (tänk på att all PubSub-data måste vara syntaktiskt korrekt XML) enligt principen i föregående avsnitt så gäller det att välja rätt verktyg.

XML är alltså en förutsättning men inom ramen för XML finns det gott om sätt att skjuta sig i foten. Det är relativt enkelt att konstruera semantiskt neutral syntax i enkla fall (tex i vos-release-fallet ovan) men i mer komplexa situationer är det lätt att operationella aspekter på eventet spiller över i syntaxen.

Ett alternativ som alltid kan uppfylla semantisk frikoppling är RDF. Resource Description Format är en W3C-standard som bla används i RSS 2.0, Mozilla XUL men framförallt används RDF och en relaterad standard OWL som syntax för beskrivning data i kunnskapsorienterade system – sk ontologier. Ontologier är vanliga inom bla flyg/rymd-teknologi där det finns behov av att skapa omfattande kontrollerade vokabulär (*exakt vilken skruv talar du om?*) för kommunikation mellan företag.

RDF/OWL medger uttryck av både modeller och instanser av modeller av data på ungefär samma sätt som UML gör – fast UML kan inte uttrycka instanser av klasser med samma detalj. En RDF/OWL-modell liknar på många sätt en dump av en relationsdatabas – inläst i minnet genom ett RDF-bibliotek kan en man ställa frågor till en modell (en uppsättning objekt och klasser) som motsvarar hur man ställer frågor till en objekt eller relationsdatabas.

Genom att skicka RDF/OWL-modeller mellan agenter uppnår man både semaniskt frikoppling och ett enhetligt sätt att modellera och representera datastrukturer som skickas mellan agenter.

Detta är föremål för utveckling men tester kommer att genomföras inom ramen för JEvent på SU.

Checklista för en ny agent

Följande saker behöver ske innan en ny agent kan tas i bruk:

- Skapa en serviceanvändare med associerad JID och lösenord
- Skapa evt nya noder och sätt subscirbers/publishers för dessa
- Konfigurera evt nya noder
- Bestäm vilka andra agenter måste ändras (tex map subscription/publishing).
- Skapa evt syntax för events
- Skapa konnandon för manuell kontroll av agenten – skapa evt formulär.
- Skriv koden för agenten!