

SUEvents

Leif Johansson <ts@it.su.se>

*Integrationsplattform baserad på asynkron
meddelandepassering*

1 Introduktion

Detta dokument utgör teknisk specifikation för SUEvents; ett system för komponentintegration baserat på asynkron meddelandepassering. SUEvents bygger delvis på ideer från UDS-projektet¹, närmare bestämt användningen av RDF som marshallingspråk samt SPOCP² som auktoritativ källa till routinginformation. Det finns emellertid stora skillnader mellan SUEvents och UDS.

SUEvents kommer att implementeras under 2005-2006 som ett delprojekt i uppfyllandet av IT och Medias verksamhetsmål för ökad driftsautomatisering.

2 Övergripande krav och Mål

SUEvents skall uppfylla följande övergripande krav:

- Meddelandeprotokollet skall vara XMPP.
- Flera accessprotokoll skall stödjas bla XMPP, SOAP, XML-RPC, RMI
- Mjukvarukomponenter skall kunna övervakas och administreras var och en för sig oavsett hur de paketeras tillsammans.
- Meddelanden mellan olika komponenter använder RDF.
- Meddelanden (eller 'Events') skall kunna sparas och spelas upp vid senare tillfälle.

1 <http://www.spocp.org/uds>

2 <http://www.spocp.org>

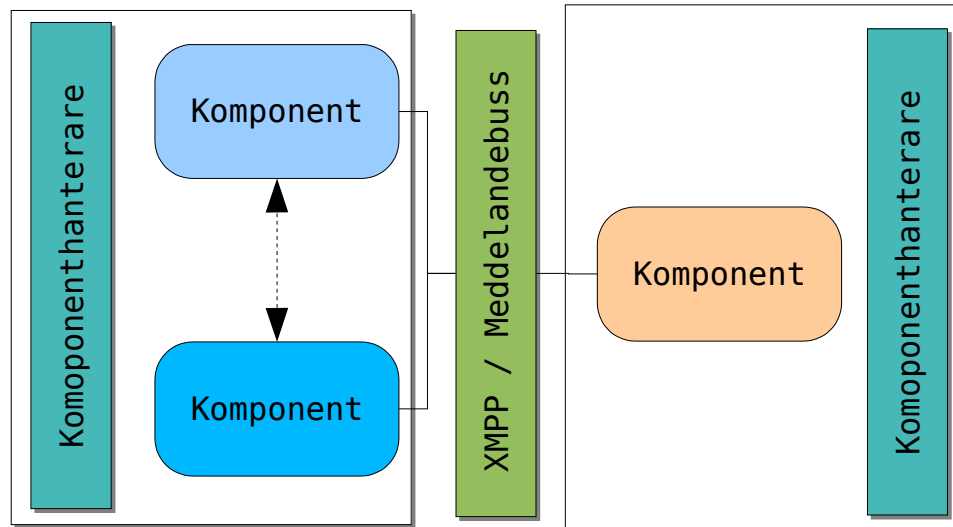


Diagram 1 SUEvents systemskiss

Det huvudsakliga målet med SUEvents är att skapa en bas för ett meta-katalogsystem. En metakatalog samordnar informationssystem till en gemensam vy som presenteras i en enterprisekatalog, i detta fall SUKAT. Det finns emellertid andra tillämpningar främst där det finns behov av att bygga gränssnitt till asynkrona processer eller processer som behöver serialiseras, tex skapa AFS-volymer.

3 Arkitektur

Den primära kommunikationen sker med XMPP-meddelanden mellan olika mjukvarukomponenter. För att hantera administration och övervakning av komponenter behöver dessa i de flesta fall någon form av logisk infrastruktur som svarar för övervakning, loggning, omstart, kommunikation, trådning, konnektivitet mot datakällor och andra informationssystem. Exempel på sådan infrastruktur är EJB, Avalon, Spring, .NET eller POE.

Komponenterna kan kommunicera på andra sätt än med meddelandepasserig inom varje komponentbehållare. Denna kommunikation kan vara både synkron och asynkron och faller utanför detta ramverk sålänge denna kommunikation inte kommer ivägen för den asynkrona

kommunikationen mellan komponenter i olika behållare.

Komponenterna själva måste agera helt asynkront och bör inte behålla tillstånd mellan meddelanden utöver konfigurationen (tex anslutning till externa informationssystem eller databaser). Mao bör hantering av ett meddelande inte påverkas av tidigare meddelanden genom information som sparas i komponenten. Denna regel (tillståndsfrihet) ger enklare komponenter (men kanske fler) komponenter. Detta ställer också högre krav på komponentinfrastrukturen.

Meddelandesystemet är baserat på XMPP JEP-0060 (publish-subscribe). Komponenter publicerar eller konsumerar meddelanden med RDF-data via pubsub.

Varje komponent motsvarar en och endast en XMPP-klient och en JID. En producerande komponents XMPP-klient behöver inte vara online hela tiden men en konsumerande komponent måste vara online hela tiden såvida inte pubsub-implementationen stödjer offline delivery.

En komponent som både producerar och konsumerar kan behöva upprätthålla mer än en förbindelse till XMPP-servern. Isåfall skall den eller de förbindelser som producerar meddelanden registreras med separata resurser och med lägre prioritet än någon av de konsumerande resurserna.

Auktorisationsmodellen i XMPP betyder att det är klienterna som fattar beslut om huruvida en annan XMPP-JID får prenumerera på en kanal. Det betyder att SPOCP-anropen sker från klienterna och inte (som i UDS) från någon central del av tjänsten. Detta har både för och nackdelar.

4 Anropsmekanismer

Ovanpå PubSub-lagret kan man konstruera flera mönster för anrop mellan komponenter.

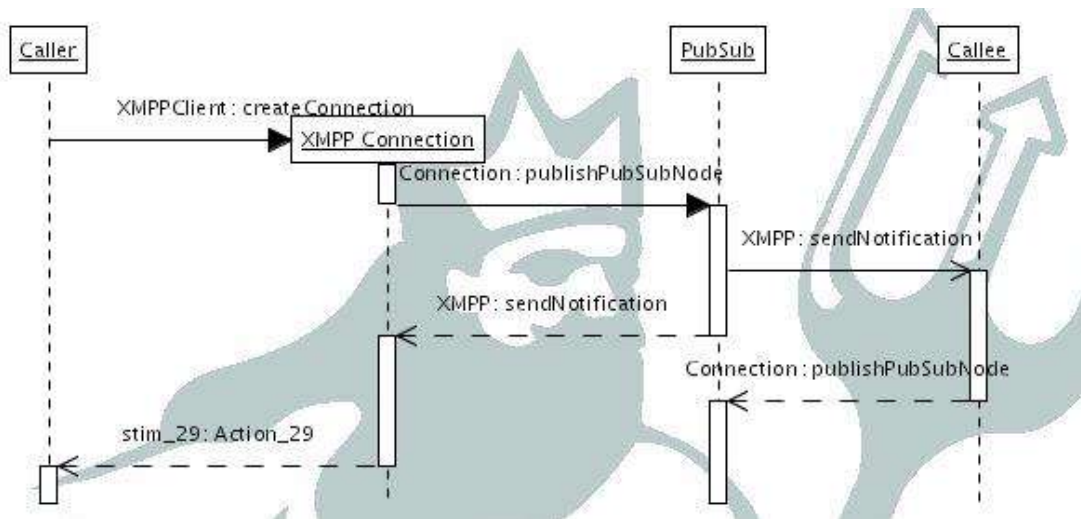
4.1 Fire and forget

Det enklaste mönstret är fire and forget; dvs vi är inte inresserade av resultatet av meddelandet, huruvida det lyckas eller hur lång tid det tar. Ett annat sätt att se på detta mönster är att anroparen inte är inblandad i behandlingen av resultatet av en operation. Detta är det mest grundläggande mönstret.

4.2 Synkront anrop med timeout

Meddelandet (Caller->Callee) publiceras. Förbindelsen som används

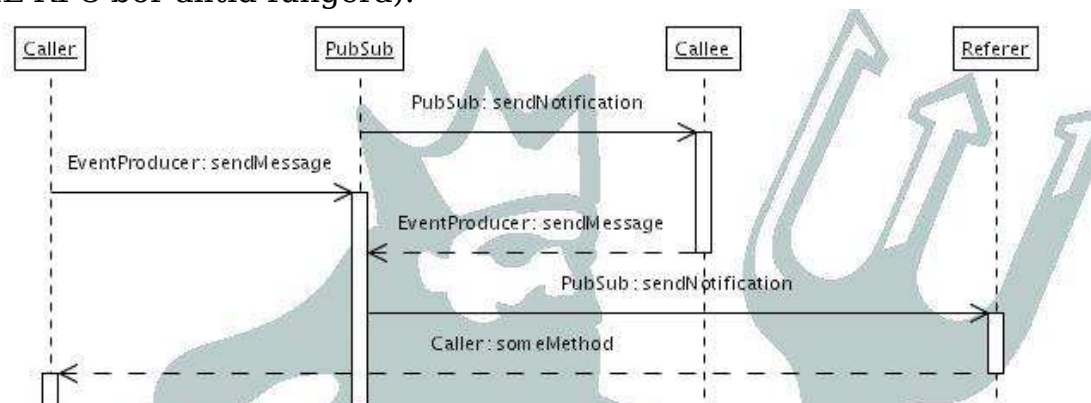
stängs inte (alternativt öppnas en annan förbindelse med en JID som får notifieringar från Callee) utan Caller väntar antingen på timeout eller på en notifiering från PubSub om att det finns data från Callee. Andra klienter kan mycket väl samtidigt vänta asynkront på samma data men det påverkar inte anropsförloppet mellan Caller och Callee.



Man kan likna detta mönster med ett vanligt synkront metod-anrop (med timeout) med möjlighet till att fler än anroparen får resultatet. Detta är i sin enkelhet ett kraftfullt mönster.

4.3 Hänvisning

I detta fall använder vi oss av ett vanligt metदानrop (vilket alltså beror på vilken komponentbehållare man använder men SOAP eller XML-RPC bör alltid fungera).



Caller skickar ett asynkront PubSub-meddelande som vanligt vilket Callee plockar upp och svarar på med ett eget PubSub-meddelande.

Skillnaden är att Caller inte längre är online och inte kan få meddelandet från Callee. Istället inkluderar Caller information om hur resultatet ska skickas med ett metodanrop (tex genom en WSDL-referens eller bara ett metodnamn). Denna information skickar Callee vidare i svaret som plockas upp av Referer. Denna komponent bryr sig inte om innehållet i svaret utan gör bara anropet tillbaka till Caller på det sätt som Caller angett. Det Referer gör är ett slags apply-funktion.

Fördelen med detta mönster är att Caller inte behöver ha en öppen förbindelse eller ta hänsyn till någon timeout. Callee behöver heller inte veta något om hur svaret ska levereras till Caller utan skickar bara med referral-informationen i svaret. I en J2EE EJB-miljö kan Caller tex vara en sessions-driven böna som använder SUEvents som ett kösystem och får notifiering via en av sina metoder när jobbet är klart.

5 Användarfall

5.1 Frontend till vos create

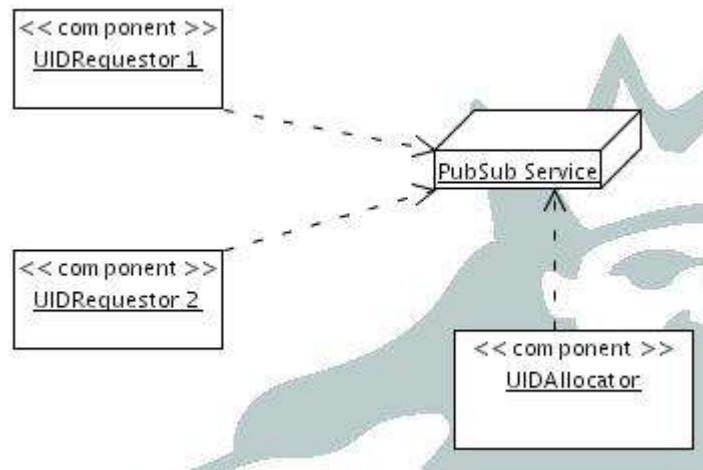
Det är av olika skäl bra att serialisera access till vos create speciellt när det handlar om att skapa AFS-volymer till användare:

- Volymoperationer misslyckas ibland och behöver då startas-om utan att den som begärt operationen behöver utföra begäran på nytt.
- Efter varje vos create user.<uid> måste man göra vos release home men två sådana operationer samtidigt är onödigt.

5.2 Allokering av användar-id

Att allokera en unik användar-identitet är en process som, i avsikt av transaktioner i enterprisekatalogen, måste serialiseras. Genom att bara deploya en komponent som allokerar id:n är unikheten garanterad

Kommunikationen mellan komponenterna liknar en request-response-modell med den skillnaden att anropet inte blockerar på svaret: För att begära allokering av ett uid publicerar en request-komponent ett meddelande till en PubSub-nod som UIDAllocator bevakar.



Meddleandet innehåller dels en information om operationen som begärs (allokering av ett uid) samt adressen till en PubSub-nod där svaret ska publiceras. När ett uid är allokerat publicerar UIDAllocator ett meddelande med det allokerade uid-värdet till den PubSub-nod som request-meddelandet innehöll.

Detta svarsmeddelande konsumeras antingen av samma komponent som gjorde begäran (om man tror eller vet att operationen att allokeras ett uid kommer att terminera på rimlig tid) eller av en helt annan komponent som tar vid där UIDRequestor-komponenten lämnade av.

6 Komponentinfrastrukturen

Kraven på infrastruktur för mjukvarukomponenter som kommunicerar med varandra genom SUEvents innebär bla att följande API:er måste existera:

- Ett XMPP API som implementerar PubSub (JEP-0060)
- Ett SPOCP API som minst implementerar QUERY.
- Ett RDF API för RDF-XML³.

³<http://www.w3c.org/TR/RDF>

Ju bättre integrerade dessa API:er är i komponentinfrastrukturen desto enklare blir det att utveckla nya komponenter.

6.1 EJB 2.1 / J2EE 1.4

I och med EJB 2.1 finns stöd för att koppla externa informationssystem till en EJB-container via JCA 1.5 resource adapters. Gränssnittet mellan en EJB 2.1-container (tex JBOSS 4.x) till SUEvents sker alltså via JCA. Det finns tre modeller för integration beroende på om EJB-komponenten är konsument, producent eller både och:

6.1.1 Konsument: MDB

En konsumerande komponent i EJB 2.1 implementeras som en MDB: message driven bean med anslutning till en JCA resource adapter som stödjer "inflow"-modellen för meddelandepassering. I deployment-descriptorn knyts en MDB via en `ActivationSpec` till en JID med resurs och prioritet. Vid aktiveringen av ändpunkten associerat med MDB:n startar RA:n en tråd i applikationsservern (via sin `WorkManager`) som öppnar en förbindelse till XMPP-servern. Denna XMPP-klient får sedan pubsub-meddelanden som den passerar upp till MDB:n på vanligt sätt.

6.1.2 Producent: CCI

Genom att använda `javax.resource.cci` med `Record`-objekt som bär RDF-datat sker anropen till SUEvents på vanligt sätt enligt "outflow"-modellen i JCA 1.5.

6.1.3 Producent och Konsument

Om samma exekveringskontext (EJB) både behöver konsumera och producera information så kan detta ske antingen genom att låta en `Interaction.execute` blocka i väntan mellan att ett meddelande publicerats och ett annat har tagits emot. Detta är dock bara lämpligt om man vet att det inte tar lång tid samt att ett svar verkligen kommer. Detta bryder i princip mot kravet på asynkronicitet i SUEvents.

Ett annat alternativ är att använda en "callback-MDB" - dvs en MDB vars enda syfte är att ta emot ett meddelande och skicka det vidare (med ett metदानrop) till den publicerande (och konsumerande) komponenten.

6.2 Perl POE

TODO

7 Behörighetskontroll

Till skillnad från UDS där SPOCP sköter både meddelanderouting och behörighetskontroll samtidigt följer SUEvents den grundläggande modell för behörighetskontroll som XMPP har nämligen att användaren själv eller i PubSub-fallet att ägaren till en nod fattar beslutet. I klassisk XMPP sker detta genom att en människa väljer att godkänna en presence-prenummeration.

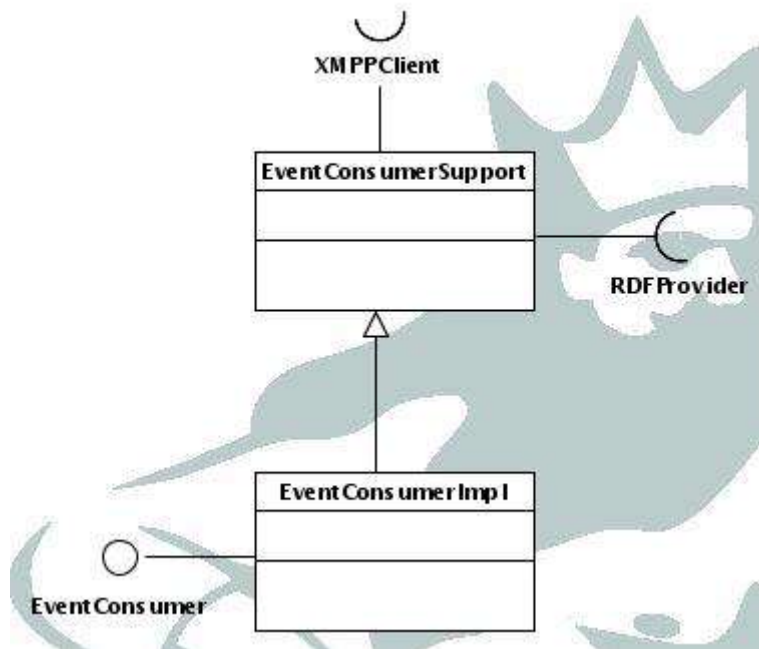
I SUEvents sker det istället genom att klienterna (komponenterna) måste implementera ett interface (se nästa avsnitt) där bks sköts. I kontraktet mellan PubSub och SUEvents ingår att klienten ska delegera detta beslut till SPOCP.

8 Objektmodell och kontrakt

En SUEvent-konsument implementerar interfacet EventConsumer som illustreras i följande kodsegment (java):

```
public interface EventConsumer {  
    public void onRDFEvent(RDF rdf);  
}
```

En EventConsumer är alltså i princip jämförbar med en JMS MessageListener med skillnaden att meddelandet som passeras är en parsad RDF-modell. För att underlätta utveckling av nya komponenter är det antagligen bäst att implementera ett komponent API i form av en support-klass som implementerar XMPP-stödet och parsar RDF.



DeploymentDiagram 1 EventConsumer

Beroende på komponentinfrastrukturen så behöver dessa relationer inte realiseras med traditionella OO-mekanismer som interface och klasser.

En producent implementerar dels interfacet EventProducer:

```

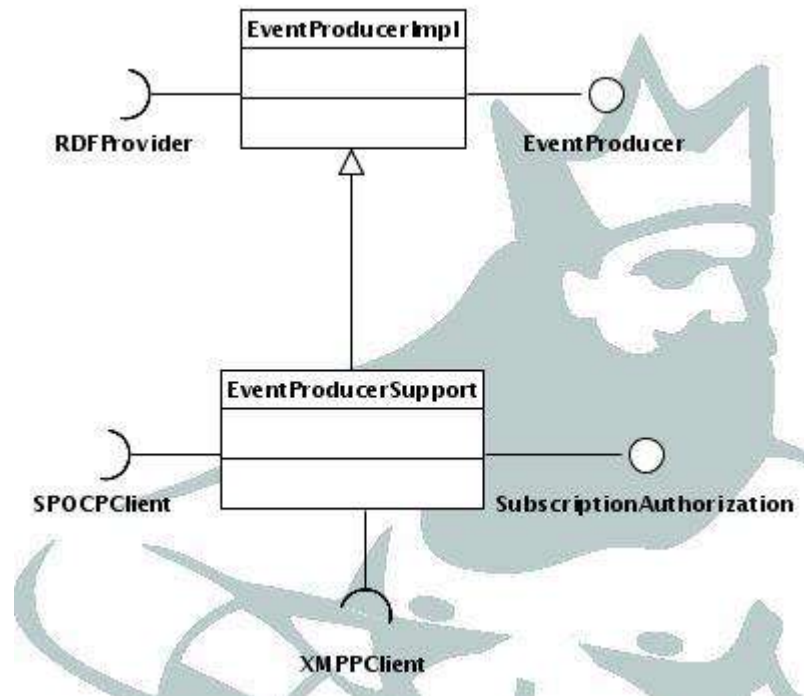
public interface EventProducer {
    public void sendEvent(RDF rdf, PubSubNode targetNode);
    ... // metoder som administrerar PubSub-noder är utelämnade
}
  
```

och dels interfacet SubscriptionAuthorization:

```

public interface SubscriptionAuthorization {
    public boolean isSubscriptionAuthorized(PubSubNode node,
                                           JID source);
}
  
```

Behovet av infrastruktur illustreras med följande deployment-diagram. De öppna socket-symbolerna betyder beroende av ett interface snarare



att ett interface implementeras av klassen. Tanken är alltså att **EventProducerSupport** hanterar behörighetsbeslut med anrop till policy-tjänsten (spocp) medan implementationen själv hanterar serialisering av meddelanden till RDF via sin **RDFProvider** och implementationen tar hand om att skicka RDF-meddelanden.