Hur den lösa kopplingen ändå blir hård

Jakten på lös koppling kan leda till att den blir ännu hårdare

BALANSGÅNG MELLAN OLIKA SORTERS KOPPLING Det brukar anses mycket viktigt att ha låg grad av koppling mellan SOA-tjänster. Men välmenande förbättringar kan leda till försämringar.

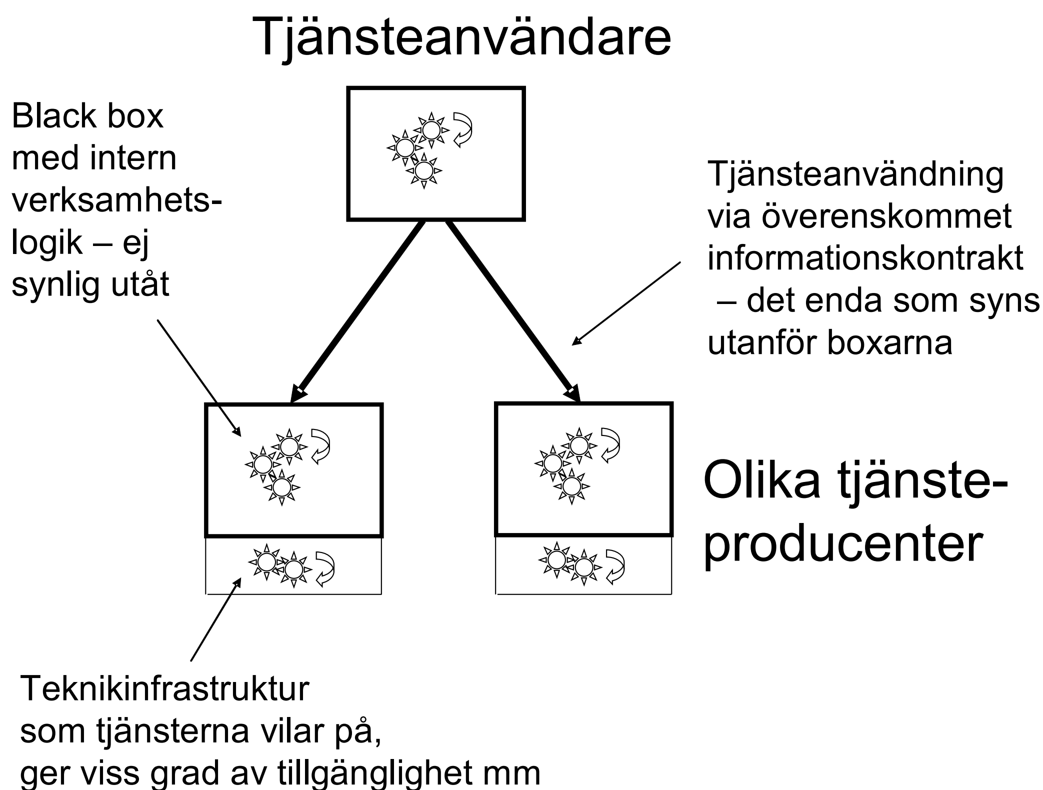
Lös koppling (”loose coupling”) brukar framhållas som en synnerligen viktig aspekt av framgångsrik SOA. Men när man granskar begreppet inser man att det finns en mängd tolkningar av lös koppling. På abstrakt nivå uttyds graden av koppling som graden av **beroende** mellan två ihopkopplade delar – i SOA-sammanhanget handlar detta om användaren av tjänsten (”consumer”) och producenten av tjänsten (”provider”).

Nu kanske du tänker: det ligger väl i sakens natur att tjänsteanvändaren är beroende av tjänsteproducenten? Utan beroende finns det ju ingen tjänsteanvändning, så varför skulle det vara så dåligt?

Varför lös koppling önskas inom SOA

SOA syftar till att förenkla. Tjänsteanvändare och tjänsteproducent ska inte behöva vara överens om myllriga små detaljer om exakt hur tjänsten utförs. ”Agreement is expensive”, har någon vis person sagt och därmed bör man hålla ner antalet detaljer som man måste vara överens om. Helt enkelt hålla nere till det minimum som behövs för att få fram önskad funktionalitet.

Här kommer också begreppet ”black box” in. Tjänsteanvändaren ska inte behöva känna till innanmätet i tjänsten och alla dess teknikdetaljer, det ska räcka med att känna till ytan på den svarta lådan - det vill säga informationsgränssnittet, tjänstens kontrakt. Det ger lösare koppling mellan tjänsteanvändare och tjänsteproducent.



En av bästsäljarförfattarna inom SOA-litteraturen, Thomas Erl, menar att det finns sju sorters koppling men att bara två av dem är positiva. De som är bra är där det funktionella kontraktet egentligen skapas mellan konsument och tjänst. De andra fem är oönskade.

I mer tekniska termer finns det många andra sorters kopplingar där det är olämpligt om de blir för hårda. Det kan vara saker som att tjänsteanvändare och tjänsteproducent inte ska behöva hålla ordning på varandras status eller läge emellan anrop (”statelessness”). Det kan vara att man inte ska använda allt-eller-inget-transaktioner (så kallad ACID) i SOA-gränssnittet, eftersom mångpartstransaktioner omöjligen rimmar med statelessness. Det kan vara att meddelanden som utväxlas ska vara grovkorniga (”coarse granular”) för att vara så självständiga och oberoende av varandra som möjligt – exempelvis att ha en hel fakturapost i ett meddelande istället för mängder av småmeddelanden med fakturahuvud för sig och varje fakturarad för sig.

Här ska vi koncentrera oss på en oberoendefaktor: **Tjänsteanvändare bör inte vara hårt beroende av att alla tjänsteproducenterna som används i just detta nu är vid liv och har bra svarstid.**

I SOA-visionen ingår att kunna plocka ihop en lösning utifrån färdiga tjänster, som byggklossar. I ett sammansatt system där man använder ett stort antal tjänster skulle man med hård koppling erhålla mycket dålig sannolikhet för att helheten verkligen är igång och svarar. Den tekniska tillgängligheten totalt sett blir usel och slutanvändarna blir missnöjda.

Asynkrona tjänstegränssnitt är ofta lösningen

Den viktigaste lösningen för att slippa denna sorts hårda koppling är att använda asynkron kommunikation mellan tjänsteanvändare och tjänsteproducent. Detta innebär att anropande tjänsteanvändare inte kräver ett omedelbart svar utan kan gå vidare med andra saker, trots att tjänsteproducenten ännu inte gjort klart sitt jobb . Tjänsteanvändaren blir därmed oberoende av att tjänsteproducenten kan svara **just nu**. Lösningen får konstrueras så att det duger att om ett svar skulle behövas, så får det komma aningen senare. Här anar ni förstås att det finns nackdelar med asynkrona tjänstegränssnitt också.

Ofta är det i alla fall funktionellt helt tillräckligt med asynkrona tjänstegränssnitt och vi vinner oberoende av tjänsternas tillgänglighet. Men då kommer nästa fråga, hur ska vi hantera om det uppstår ett fel eller undantag?

Ett bankexempel

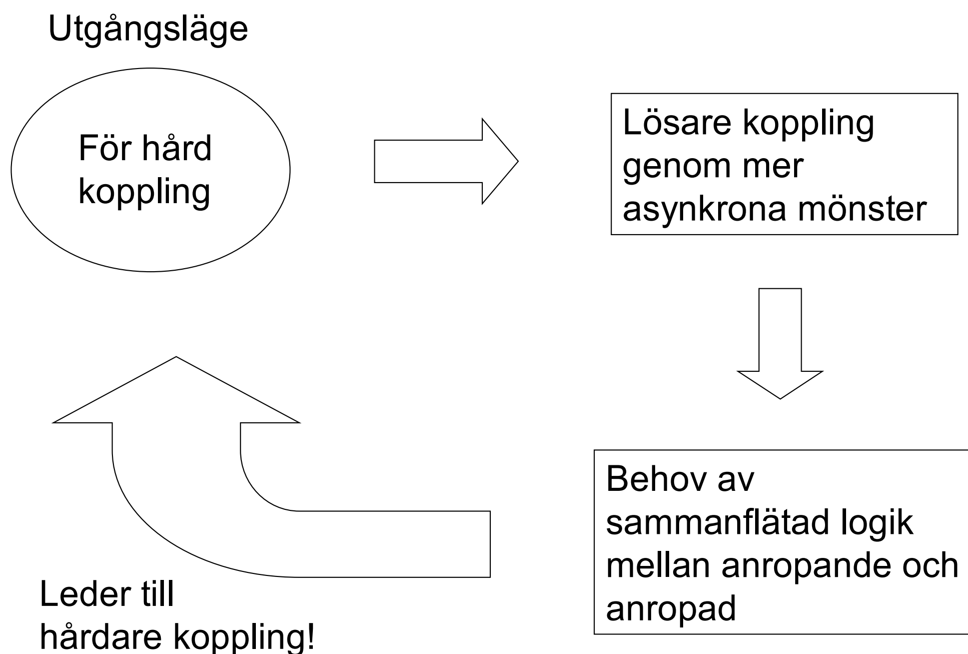
Säg att vi ska göra ett bankuttag. Om vi utformar det synkront får vi omedelbart reda på om det är slut på pengar på kontot eller om bankens kontosystem har tekniska problem. Vi kan meddela en slutanvändare om situationen omedelbart. Ifall det rör sig om system utan användargränssnitt kan ändå anropande system direkt fatta bra beslut grundat på väluttänkta returkoder.

Om istället ett asynkront scenario används så kan man tänka sig att anropande system först frågar om ett uttag på 20.000 medges. Anropssystemet sysslar med annat ett tag, och får sedan ett asynkront meddelande om att det går bra. Därefter skickas en uttagsbegäran som ett asynkront tjänsteanrop. Anropssystemet måste anta att detta ska gå bra eftersom det frågat först. Det använder nu pengarna till ett börsköp. Men om nu det var så att det faktiskt

kommit in ett bankomatuttag på 8.000 under mellantiden så räcker inte saldot till. Det måste skapas ett asynkront protestmeddelande. Därefter måste anropssystemet försöka krypa ur börsaffären. Som också är asynkron till sin karaktär...

Kanske är kontoexemplet lite förenklat och kanske kan man utforma banklogiken bättre, men det viktiga är det principella resonemanget att logik inuti tjänsteanvändaren och logik inuti tjänsteproducenten lätt blir synnerligen sammanflätad i asynkrona sammanhang. Parterna blir ordentligt sammanflätade redan i normalflöden och ännu mer i flöden för undantagshantering. Sammanflätat är inte lika med oberoende.

Det har uppstått en ny slags hårdare koppling (mellan logikinnanmäten) som resultat av en ansträngning att skapa mindre hård koppling (övergå från synkront till asynkront).



Det finns designteori kring hur man skapar asynkron undantagshantering, till

exempel mönstret med långa verksamhetstransaktioner (”long running transactions”) men det är viktigt att inse att detta är ett svårt ämne. Det kostar verkligt mycket tid och pengar att designa både denna verksamhetslogik och teknikfelslogiken. Dessutom är testning av asynkrona mönster arbetsam. Med andra ord: risk för fel i felhanteringen!

Lösningar

Det måste alltså finnas en balansgång mellan olika sorters lös koppling. Att skapa en optimerad lösning innebär att man klarar av att kompromissa. Detta gäller för övrigt även andra aspekter av lös koppling än de två som tas upp här.

Vad ska man då göra åt saken, mer konkret? Här kommer några lösa förslag:

- Utgå från verksamhetens behov när du analyserar frågan om asynkront/synkront. Vilket färskhetsbehov finns på informationen? Omedelbart synkront (såsom en lagerfråga), ett dygn (såsom för fakturaöverföring)? Kan man kanske replikera register till den andra SOA-domänen (såsom till prisuppgifter)?

- Gör inte ”okynnesuppdelning” i alltför små SOA-domäner. Vissa gånger är det bättre att hålla ihop logik som hör till ofta förekommande användningsfall. Logiken hamnar då på insidan av en lite större SOA-tjänst. Inuti en sådan black box är lös koppling inte alls lika nödvändig. De förenklande allt-eller-inget-transaktionerna kan vara tillåtna.

- I de fall som man vet att tjänsteproducenten säkert kan garantera hög tillgänglighet och kort svarstid kan synkrona mönster övervägas.

- Om du märker att asynkrona mönster behövs i stor skala i din lösning, gör åtminstone en realistisk budget för det stora arbetet att utforma och testa alla långa verksamhetstransaktioner.

Lär mer om moln (och även knytningen till SOA) på kurs

För- och nackdelar med de olika molnen, liksom SOA, integrationslösningar och migrering gås igenom noggrant i tredagarskursen ”Cloud computing – migrering och integration” som ges 1 – 3 december 2009. Se Dataföreningen Kompetens, www.dfkompetens.se