Skaffa eller skrota ESB?

Borde vi införa en ESB (Enterprise Service Bus), strunta i det eller kasta ut den vi har? Liknande resonemang gäller för övrigt även andra plattformsval!

2015-10-07: Sven-Håkan Olsson

Skaffa eller skrota ESB? **En ESB (Enterprise Service Bus) kan ofta ge stor nytta men andra gånger ger den mest nackdelar. Skaffa, skrota eller behåll?**

Under ett antal år har det varit populärt att införa ESB:er. Ofta har det varit ett sätt att få kontroll över en mängd integrationer mellan applikationer. Ibland var det en del i en SOA-satsning. I vissa fall har ESB:n även använts för kommunikation inom en enda applikation eller som client/server-förmedlare (numera även app/server-förmedlare, liksom webjavascript/server-förmedlare).

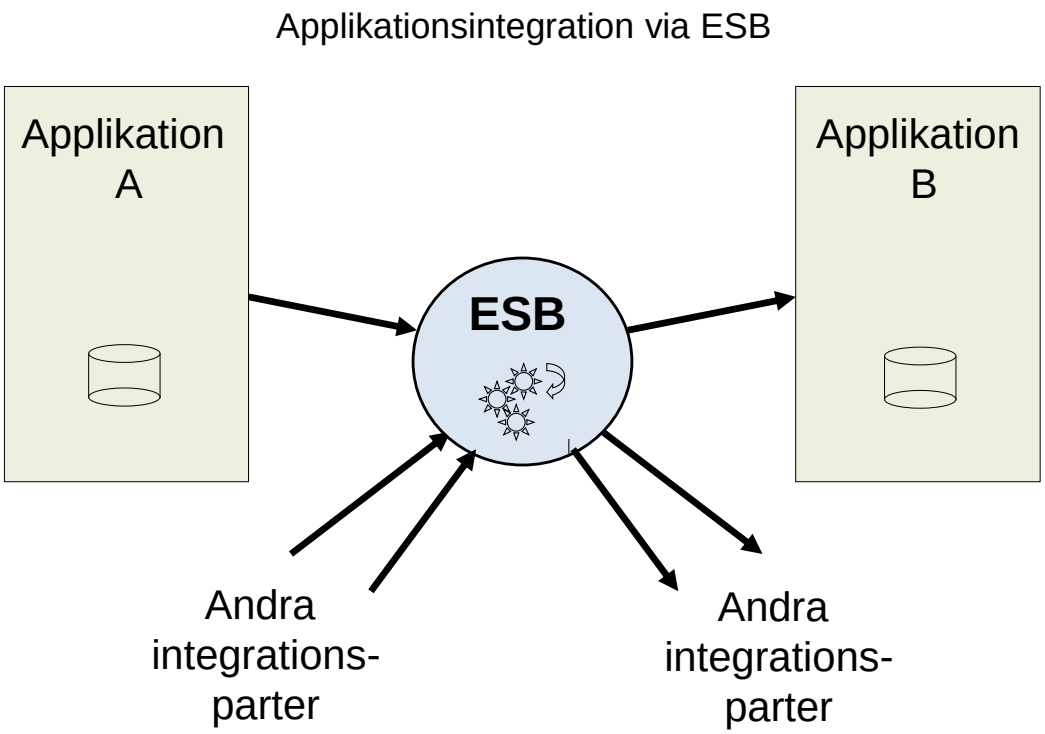
Det finns ett antal fördelar och nackdelar med ESB:er. Jag kan tycka att man ibland har sett lite för lättvindigt på nackdelarna och närmast av slentrian valt ESB-spåret. Därmed inte på något vis sagt att ESB alltid måste vara dåligt!

Man kan faktiskt utsträcka beslutsresonemangen till många andra val av stora ramverk, middleware, plattformar och speciella arkitekturer. Ibland tycker jag inte man går igenom för/nackdelar tillräckligt klarsynt innan man fattar beslutet. Så resonemangen nedan kanske kan hjälpa till även vid sådana beslut.

Innan ESB:erna blev kända använde man ibland produkter för EAI (Enterprise Application Integration). En del av dem har en lång tradition och det fanns exempel på "transaktionsväxlar" eller liknande redan på 70-talet. EAI-produkter och ESB:er har en del gemensamma drag, bland annat brukar de erbjuda behändig formatkonvertering och enkel avtappning av information för nya ändamål. Men förenklat uttryckt var EAI optimerat för stora dataflöden medan ESB ofta är bättre på kortare anrop.

Det kan också vara värt att nämna att lösningar för ETL (Extract Transform Load) som normalt används inom Business Intelligence också kan erbjuda delmängder av funktionaliteten hos ESB:er och kan fungera bättre för något visst syfte. EAI-lösningar kan

också ha fördelar även idag. Framförallt tänker jag på de prestandaproblem som ESB:er ibland ger när dataflödena är stora och inte passar in i den vanligen rådande anropstanken hos en ESB.



En liten paradox är att ESB:er alltid brukar ritas som "nav", precis som jag har gjort i figuren ovan, medan en "äkta buss" kanske borde ritas annorlunda – utsträckt geografiskt och med likvärdiga parter anslutna?

Hur ska man då tänka kring av välja att ha ett ESB-nav eller inte? Nedan följer några plus- och minusposter som ofta återfinns inom verklighetens ESB-införanden (utan prioordning).

Några fördelar

- Ge kontroll över vilka flöden som existerar, motverka "inter-application spaghetti". Tillföra rättighetsstyrning och loggning.
- Behändig formatkonvertering ”på vägen”.
- Persistent kö med leveransgaranti. Ibland bra hantering av gammaldags transaktionsfiler.

- Kontrollerad kommunikation en-till-många genom "publish/subscribe".
- Återanvändning av existerande flöden, tjänster, sammansatta tjänster och gemensam verksamhetslogik som tillgängliggörs via ESB:n. Till exempel kan det gå lätt att koppla in en ny mottagande applikation som plötsligt behöver information som redan finns i navet (detta underlättas även av förra punktens subscribe).
- Kunna ta ut affärsnyckeltal, användningsstatistik eller debiteringsinformation eftersom ESB:n är en central punkt där en stor del av organisationens data passerar igenom.
- Centralt kunna införa verksamhetslogik, antingen via grafisk programmering med peka-och-klicka, processpråk (BPMN, BPML, XLANG etc) eller traditionell programmering.
- Ibland kan man få fördelar av att använda en generell datamodell (kanonisk modell) inne i navet och alltid gå via den. Även här har man nytta av navets inbyggda konverteringsstöd. [1]
- Vara en tydlig plats att publicera tjänster på och underlätta att hitta dem via (tjänstekatalog).
- Kunna ge ett oberoende från underliggande kommunikation. Exempelvis kanske exakt samma flöde skickas till olika parter antingen via ftp eller MQ. Detta val är vanligen en konfiguration som kan ändras utan att ändra på andra definitioner eller på programmering i navet.

Några nackdelar

- ”SPOF – Single point of failure”. Hela organisationen kan stanna, dels om navet tekniskt sett kraschar, dels om navpersonalen råkar göra fel. [4]
- Komplexiteten kan bli avsevärd på grund av att navet behöver en avancerad struktur för att vara flexibelt nog. Komplexiteten ökar även vid klustring. Blir alltså lätt att göra fel i driften. [4]
- Naven brukar ha en rejäl kunskapströskel för att någon ska kunna utveckla lösningar. Det tar minst ett år att bli tillräckligt kunnig för att sjävständigt arbeta med navet.
- Eftersom det endast brukar vara några få personer som kan tillräckligt om navet (framförallt på utvecklingssidan, men även på driftssidan) blir de personerna en trång sektor när förändringar behövs. Visserligen kan man snabbt och effektivt återanvända tjänster som redan finns i navet, men ofta behöver man ändå lägga till något fält eller göra någon ny liten konvertering. Leveranstid på ett halvår kan lätt uppstå även för enklare ändringar.
- Ofta placeras de navkunniga i en central grupp (ibland kallat ICC, Integration Competence Center) vilket är en bra idé för att odla kunskapen om vald ESB. Men

här kan uppstå prioriteringsproblem. Om exempelvis en av affärsgrenarna anser att de har extra bråttom med en ny funktion så har de inte själva makt över situationen utan kan behöva gå högt upp i ledningen för att ändra på prion inom ICC.

- Kostnaden för licenser och servrar kan bli avsevärd eftersom man brukar behöva minska SPOF-risken genom klustring och liknande (vilket i och för sig inte hjälper mot operatörsmissgrepp). Dessutom behöver man ett antal test- och utvecklingsservrar vilka också kan kosta mycket i licens. Ibland krävs också dyra databaslicenser.

- Versionering av konfigurationer och av grafisk programmering har inte alltid varit enkel. Både vad gäller att spara undan versioner så man vet vad som gällde vid vilken tidpunkt, och att snabbt kunna backa ett misslyckat ändringsinförande. Liksom att sammanhållet och enkelt föra över ett versionspaket från testmiljön till drift. [3]

- Att tänka ut undantagshanteringen för en ESB är mycket viktigt och kan bli för simpelt utfört, varvid datakvalitetsproblem uppstår. [5]

- Navet tillför ett extra skikt som kan ge prestandaförlust och fördröjningar vid online-behov.

- Navet tillför ibland prestandaproblem då riktigt stora flöden ska passera.Vissa verksamheter behöver information av gigabytekaraktär, så hundratusentals transaktioner ska snabbt bearbetas satsvis – men ofta är naven snarare optimerade för korta meddelanden än denna sorts last. Man kan därför behöva införa segmentering för att få kortare meddelandeströmmar så att inte navet ska storkna. Sådan segmenteringslogik kan kosta mycket att utveckla eftersom den har bäring även på undantagshanteringen.

- Definitioner av verksamhetslogik och formatkonverteringar inne i navet riskerar få otydliga ansvarsägare. Ibland hamnar definitioner av praktiska skäl inne i navet trots att de ansvarsmässigt tillhör en annan "black box". [2]

- Precis som andra mjukvaruprodukter så har ESB:er begränsad livslängd. Kanske bestämmer sig leverantören på att satsa på en annan häst eller lägga ner. Det gäller då att hinna få god utväxling av navet innan man måste byta. Lösningar inom Öppen källkod kan vara bättre i vissa fall, men kan istället ha problemet att communityn kring öppen-källkods-projektet blir inaktiv.

Gör en viktad lista för/emot

Som så ofta kan det vara fruktbart att lista den här typen av faktorer mer specifikt för just sin egen organisations situation. Vad är extra noga för oss? Sedan kan man sätta en ungefärlig "vikt" på varje för/nackdel samt normera med avseende på antalet faktorer. Ut kommer en grov indikation på om man bör slå till eller inte!

En sådan övning tenderar dessutom att ge en mycket större respekt för det arbete som behöver utföras i samband med integrationer, med eller utan ESB. Som nämnts i inledningen kan den typen av beslutsunderlag användas även för många beslut gällande andra stora plattformar än ESB:er.

Om man nu inte väljer nav

Slutligen, några tips om man nu skulle välja att INTE använda en ESB:

- Utgå från alla-till-alla-kommunikation. Men man kanske måste skapa striktare rättighetsstyrning för vissa integrationer. Ger spaghetti-risk och detta måste man kanske kompensera genom annan systemutveckling, specifik loggning eller genom användning av någon enklare mellanprogramvara. Det har vuxit fram produkter inom API Management och liknande som är klart intressanta – se bara upp så en sådan produkt inte blir lika komplex som en hel ESB. [6]
- Skapa en befattning som kallas Integrationsmissionär eller liknande, ett slags "ICC light". Personen/personerna ska aktivt bistå projekten med mallar, standardiserings-ansatser och förslag enligt tipsen i den här artikeln eller annan ”best practice”. Befattningen bör ansvara för att det alltid finns en hyfsat aktuell integrationskarta.
- Skapa formatkonverteringar, sammansatta tjänster eller annan integrationslogik som behövs genom traditionell systemutveckling istället för inne i nav. Fördelen är att helt vanliga programspråk och välkända plattformar kan användas så man slipper kompetensproblemet som ESB:er för med sig. Dock måste man förstås fortfarande förstå sig på vilka integrationsproblem i sig som kan uppstå.
- Om asynkron kommunikation önskas ska man kanske införa en enklare (interoperabel) kö-produkt, några kö-tabeller eller RSS/Atom. Flera av de stora molnplattformarna erbjuder också olika sorters köer och mönster för "uppdatering strax".
- Förvånansvärt ofta finns det stora behov av att hantera gammaldags transaktionsfiler. Dessa ska överföras, ofta via ftp, men här kan man ha nytta av något filöverföringsverktyg så man slipper skriva trassliga script för ändamålet. Som nämnts ovan kan ETL ibland också komma ifråga.
- Betona avtal och beskrivningar. Hitta sätt att sammanställa vilka integrationsgränssnitt som finns. Använd en tjänstekatalogprodukt (dock återigen risk för komplexitet), en välunderhållen webbsida eller en intern wiki.
- Skapa mallar för hur integrationer ska utföras och dokumenteras. Se till att mallarna följer med i tiden och att de följs.
- Loggning på båda sidor om integrationsgränssnitten är viktigt även i vanliga fall men här blir det extra prioriterat eftersom navens inbyggda loggmöjligheter i detta fall inte erbjuds.

Lästips

Några av mina tidigare trendspaningar (av lite olika ålder) handlar om näraliggande ämnen. Mycket känns fortfarande aktuellt, t ex:

[1] Skapa en generell informationsmodell?
[2] Bör den svarta lådan vara grå?
[3] Versionshantera utspridd logik!
[4] När hög tillgänglighet inte blir hög
[5] Till den som sitter med klistret
[6] ESB:n är död – leve ESB:n!

Sven-Håkan Olsson sysslar ofta med en läsplattetjänst för bolagsstyrelser och nämnder. I övrigt är han oberoende konsult som särskilt arbetar med att kombinera verksamhetsnytta med teknikhöjd. Han har en lång karriär bakom sig sedan 70-talet som it-konsult (applikationsarkitektur, systemdesign, programmering, reviewer, utredningar, kursledning, upphandling). Sven-Håkan är medgrundare till KnowIT där han också var teknikchef 1990 - 2003. Han har flera gånger utsetts till en av "Sveriges topputvecklare" av Computer Sweden. Sven-Håkan håller regelbundet kurser åt Dataföreningen. Läs gärna mer på hans blogg definitivus.se samt på styrelsemöte.se.

Sven-Håkan Olsson