BizTalk ESB Toolkit Architectural Notes

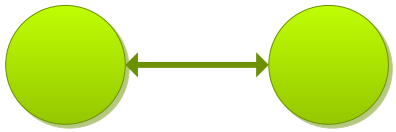
Prepared by: Joe Simon, Lead Architect, Stott Creations
August 18, 2012

The term Enterprise Service Bus (or ESB) as a software architectural model has become extremely well known as services oriented architecture becomes more widespread. Although the term ESB has morphed quite a bit and has a number of different interpretations, it's a vital piece of the puzzle in building a scalable and more maintainable messaging and services-oriented infrastructure.

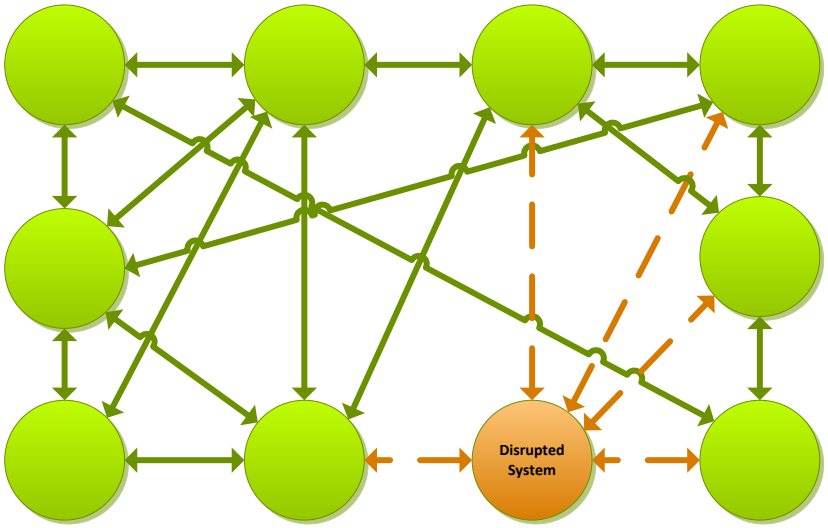
Failed Integration Attempts

As usual in the world of software development, we found a workable and efficient model (ESB) only after exhausting all other possibilities. It became obvious after a number of decades that companies were spending an inordinate amount of time in integration – trying to move data between different applications that were deployed separately, and often implemented in a very custom way. Over time, it's been estimated that up to 70% of the code in corporate software systems has been tied up in integration.

The first attempt at solving this problem was with point to point (P2P) integration solutions. For example, take the following map between a SalesForce CRM portal and a SAP installation. Both are sharing data using the services exposed in their APIs:



So far so good. But what happens when the interfaces between these independent entities change – such as when the latest version of the CRM software changes? Because all the integration code is tightly coupled in that link between the two entities, immediately the data flow will break. The cost of managing and maintaining these links escalates once multiple systems are bound together in this way. The complexity of monitoring and handling problems in the system below, especially where one piece changes, quickly rises to the point where the network of tightly bound systems becomes a cats-cradle as in the diagram below.



A system like the above – even if some or all of these services are hosted in the cloud – will experience serious problems that gradually escalate in intensity:

- **Brittleness.** Maintenance and operations becomes a constant drain and versioning makes deployments and support a recurring nightmare.
- **Administration.** It's still difficult to monitor and get visibility into what one endpoint is doing
- **Wasted effort.** Development teams have to write redundant functionality into different endpoints to handle the same type of processing.

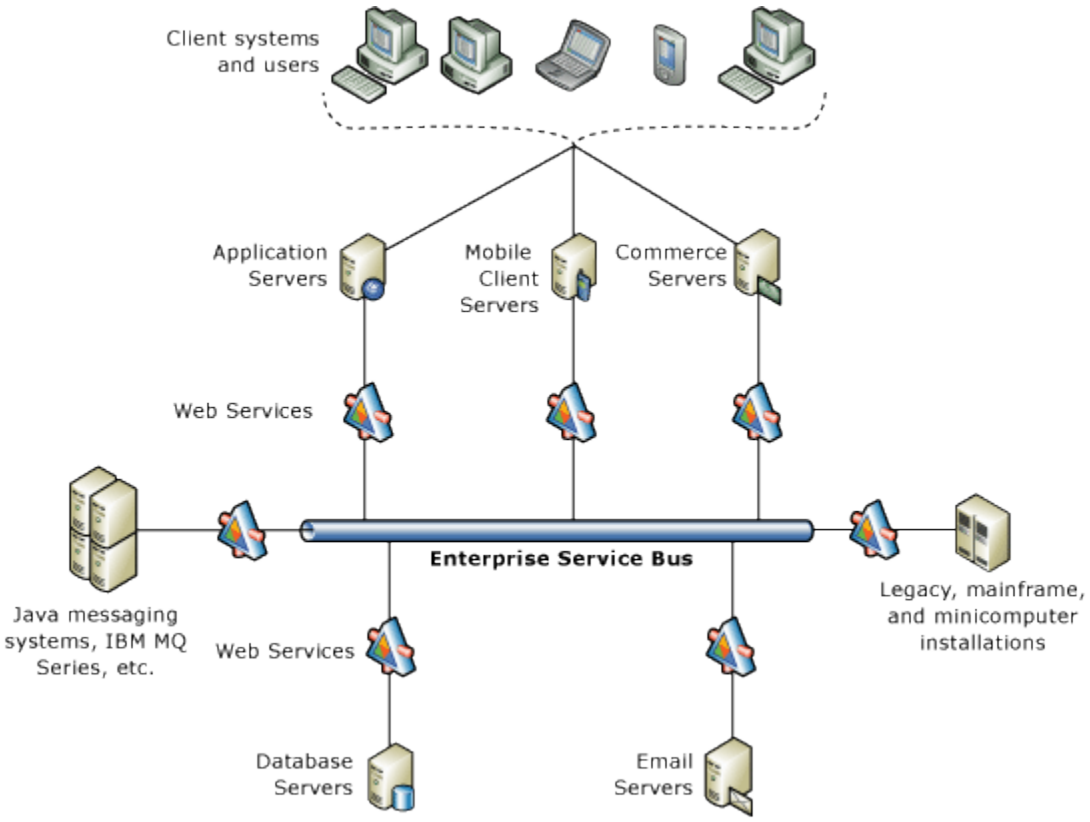
To try to resolve these difficulties, software architects and developers experimented with different Enterprise Application Integration (EAI) models. This solved one of the biggest problem with P2P – the lack of a centrally organized and managed system – but some faults remained. In particular, the endpoints of these services remained hardcoded, which led directly to cascading failures when an underlying service changed.

Services oriented architecture (SOA) was also used widely and met with varying success. This approach treats applications as black boxes, dealing mainly with the input/output feeds from the apps themselves. Multiple services could be stacked together with this approach to deliver higher levels of functionality. SOA, while conceptually very sound, in many cases however ended up implemented on a level not much better than P2P systems – and became over time in many installations just as monolithic and tightly coupled as P2P, but with the additional burden of being harder to diagnose/troubleshoot faults.

A New Model

Software engineers began to look at the problems of integration of these systems from a new angle – that of the original complex human integration system, highways. Civil engineers would never design a point to point road system to allow people to travel from Oregon City directly to Newberg, Oregon, for example. Instead, they would design an onramp – a receive port - in Oregon City from an existing freeway, and build another offramp (a send port) at Newberg. Such a design could be robust and extendable as cities would have new service needs, and changes to one endpoint would not disrupt the entire system.

As integration patterns and frameworks such as BizTalk became more commonplace, the concept of an Enterprise Service Bus began to be applied, as we can see below:



Such an ESB pattern turns the traditional problem of binding together applications on its head. What if we thought of applications not as the traditional cats-cradle of nested, complex interdependencies, but instead separated them out into as a series of processes? For example, creating a new order in a system – passing from SalesForce through to an order fulfillment system and out as an invoice – could be seen as a single workflow.

By thinking in terms of a series of steps in an itinerary, the ESB model creates a centralized, loosely coupled, dynamic integration layer. The ESB implementation in BizTalk handles data transformation, protocol conversion, and dynamic routing – abstracting and centralizing all that code away from the application space - and each application can easily present what is happening to users in a particular

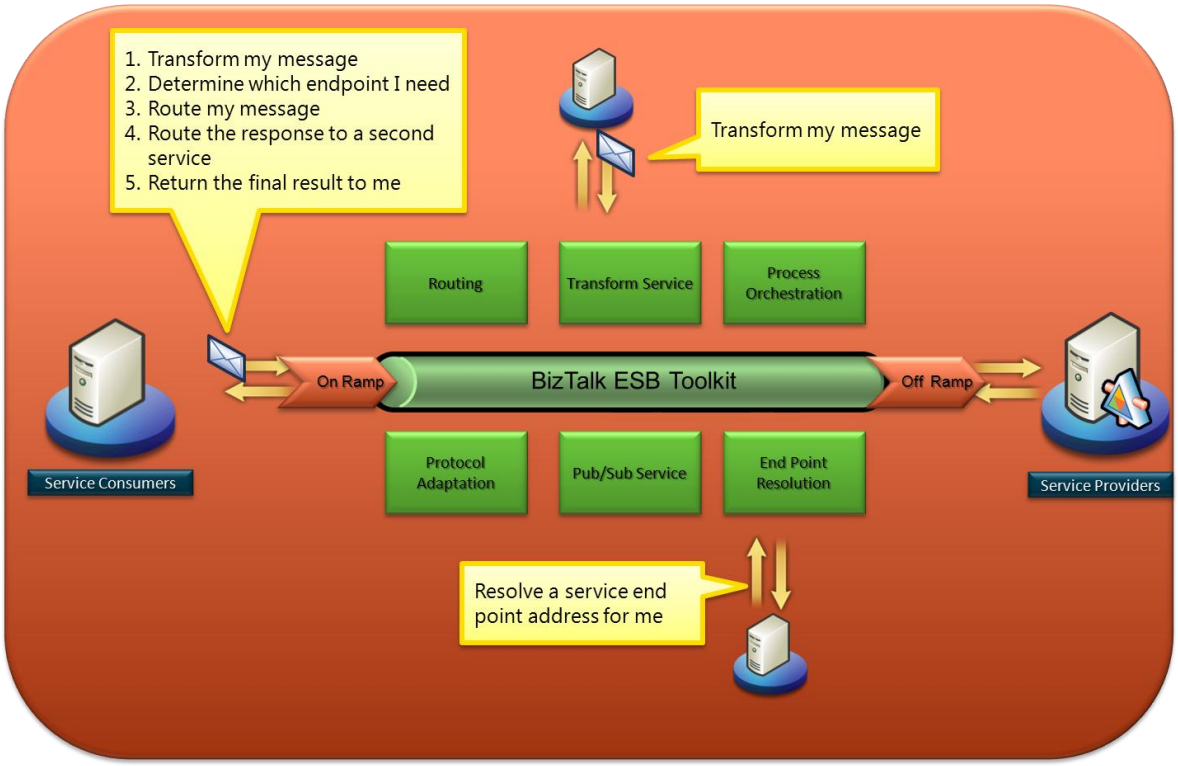
interaction. The end result is lower maintenance, less redundant code, and lower operating costs and fix downtime.

In the event of a change, such as a mandated change to the message format or a new version, there's much less disruption with this model. The ESB layer can modify the configuration without changing the individual applications themselves.

The ESB Implementation Within BizTalk

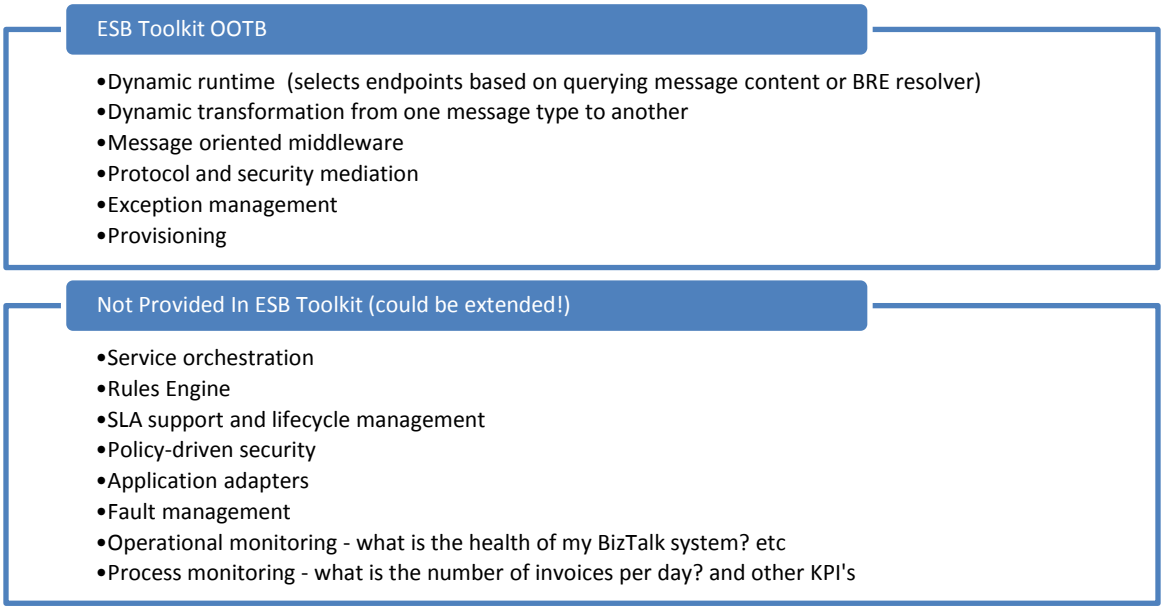
The ESB concept took hold in the early 2000's as SOAP/WS-* specifications matured. As services grew up, ESB was seen as a cornerstone of SOA. It provides a messaging fabric, a common set of services, and a runtime environment upon which developers can build out services to connect applications.

In an actual BizTalk implementation, this works much like the following:



The ESB Toolkit provided with BizTalk, which appeared in 2005, met most of the key requirements with a ESB implementation. And, most importantly, the BizTalk ESB implementation was still the familiar publisher/subscriber model, so adopting it wouldn't require learning a new set of skills. It led developers down a path of creating applications using broadly accepted best practices for BizTalk. For example, a developer could create a traditional BizTalk app using static, hardcoded endpoints – but this can't be done with ESB components, which requires a dynamic resolution.

It's also important to understand that the ESB Toolkit is simply a starting point. By a strict definition, it's not fully functional out of the box. Instead it provides a base set of ESB components that must be extended on to fit the definition of ESB:



It would be possible for developers to create their own message-oriented middleware, perhaps using some combination of WCF, WF, or AppFabric. However, in trying to implement the key features already inherent in BizTalk - scalability, admin tools, high availability, traceability, and state management - they would have to overcome from scratch the same obstacles already surmounted in BizTalk.

References

- BizTalk Deployment and Best Practices Whitepapers. http://msdn.microsoft.com/en-us/library/ee317854(BTS.10).aspx
- TechEd Video, Jon Flanders 2010, “Using Microsoft BizTalk ESB Toolkit and Integration Patterns to Improve Business Agility” http://channel9.msdn.com/Events/TechEd/NorthAmerica/2010/ASI309
- BizTalk Server Best Practices Analyzer 1.2, http://www.microsoft.com/en-us/download/details.aspx?id=15963
- BizTalk benchmark monitoring article, http://blogical.se/blogs/mikael/archive/2009/11/26/benchmark-your-biztalk-server-part-1.aspx
- BizTalk 2009 Performance Optimization Guide, http://www.microsoft.com/en-us/download/details.aspx?id=18238
- BizTalk Best Practices article, http://geekswithblogs.net/VishnuTiwariBlog/archive/2011/05/14/biztalk-server-best-practices-again.aspx

Appendix - BizTalk Best Practices

Best Practice	Category	Benefits and Explanation
Extensible application that is loosely coupled	BizTalk configuration	This is enforced much more consistently with the ESB model.
Use separate Windows groups for BizTalk groups instead of one central service.	BizTalk configuration	This is done in setup and avoids a single dependency point in case of failure and improves security. For example, use svcbtsadmin to install and config Biztalk to setup BizTalk, and later disable. Also change the SQL jobs owner to a separate account for security purposes.
Passwords should never be stored in cleartext, and all references to explicit filepaths removed or minimized.	BizTalk configuration	A basic security precaution.
Publish the external schema, not the orchestration.	BizTalk configuration	
Monitor the size of databases and tables in BizTalk – these can bloat in a disastrously short period of time.	Database backend	
Keep track of the Physical Disk objects using performance monitoring tools such as perfmon	Database backend	All the physical disk counters are important to monitor in BizTalk, especially Avg Disk Sec/Read, /Write, etc. Anything >20 msec is considered slow. Multiple tuning and BizTalk health dashboarding tools exist, and should be evaluated and put in place where bad performance is suspected. For example, we've used and recommend the Best Practices Analyzer from Microsoft, and the CodePlex benchmark tuner for BizTalk 2006/2010.
Use dynamically generated itineraries versus attached itineraries in messages	General BizTalk best practice	This removes one source of hardcoding in the design.
In design, use the ESB Toolkit as the starting point – but not the endpoint.	General BizTalk best practice	The ESB portal as delivered is to expose the capabilities of the ESB toolkit data and their relationships but is not a production grade site as-is.
All two-way services into BizTalk produce a response, either an exception or an acknowledgement.	General BizTalk best practice	
Calls to request/response web services use asynchronous callback patterns.	General BizTalk best practice	
Messages are validated against the schema for use case requirements.	General BizTalk best practice	
Just Enough Logging (JEL) - scale back logging post-go live to sane levels	Logging and exception handling	Logging every message is demanding on database and storage resources, and over time becomes less valuable. Post-deployment administrators only care if BizTalk is running and if they have the ability to drill in on errors.
For longrunning processes, users should have a way to inspect progress to date.	Logging and exception handling	This is done through dashboarding, such as SharePoint.
Do not enable tracking for ports and orchestrations, and minimize the tracking of MsgBody. Do enable tracking on Biztalk server hosts.	Logging and exception handling	As a general, the greater the amount of logging, the less said logging is used. Minimize and target the tracking and logging you actually need.
Use standard versus custom components.	General BizTalk best practice	Once a team goes down the path of constructing custom components to fit their view of what an ESB is, the solution instantly becomes more difficult to understand by every member of the development team, monitor, and maintain.
Avoid complex deployment scenarios using shared custom core libraries	General BizTalk best practice	As a rule of thumb, always exhaust all possible scenarios using out-of-the-box (OOTB) components to solve an issue before moving to custom code. Most scenarios can be accomplished by leveraging the samples provided, at the very least as a starting point. Shy away from starting from scratch. This also eases deployment and

Best Practice	Category	Benefits and Explanation
		maintenance (no custom core components means nothing in the config files. Everything can be deployed in a comprehensive package)
Write tightly focused, small components	General BizTalk best practice	This promotes solid designs and reusability. Break up development in small units of work and string these solutions together in the itinerary.
Choose config over coding	General BizTalk best practice	Always favor config changes over code changes. Compiling and retesting entire libraries for minor tweaks is bad practice.
Name your port operations so you can easily identify them in orchestrations	Orchestrations	This helps in troubleshooting issues later.
Using SQL sprocs versus ADO.NET for CRUD operations within an orchestration	Orchestrations	
Promote only those properties that are required for routing or tracking. Use the distinguished property for expression shape usage.	Orchestrations	
Naming standards follow a set convention.	Orchestrations	Multiple examples of a well-thought out naming practice exist; all are fine if followed consistently.
Use XSLT as much as possible in mapping.	Orchestrations	XSLT that use inline script or referenced assembles are inherently slower.
Where .NET objects must be referenced, don't use them directly but use external assemblies instead.	Orchestrations	
Minimize or eliminate the use of functoids	Orchestrations	This is especially true of database functoids, which are known performance killers.
Eliminate orchestrations for messaging only patterns.	Orchestrations	
Always try to design orchestrations with Direct-Bound ports.	Orchestrations	
No expression shape contains an excessive amount of code that could be in an external configuration.	Orchestrations	
Resolve addresses and maps at runtime using ESB toolkit resolvers such as the BRE resolver versus static endpoints	Resolver components	This means that addresses and maps can change without resetting host instances, and the same workflow can call different versions of a service - getting the most out of SOA architecture. Itinerary can be changed and new services added without code changes or redeployment.
Your endpoints should always be resolved/configured via BRE over UDDI	Resolver components	Power users often want to change endpoints and map types; the BRE as a resolver engine is the most powerful and flexible engine available. Comparatively UDDI both lacks intelligence and is has a complex interface configuration.
Always transform messages into a custom canonical schema.	Send/Receive Ports	This reduces the number of maps needed. Receive ports will use this one schema to validate all inbound XML.
All request/response ports that are exposed as web services are equipped with a SOAP fault message.	Send/Receive Ports	
The solution is tested with real-world data for the source system, using user accounts with identical permissions as in production.	Testing	

Best Practice	Category	Benefits and Explanation
Use maps for simple scenarios as a general rule.	General BizTalk best practice	The map itself is good for simple scenarios; but can be difficult to apply to complex scenarios. Like going from C# to C++ - in C++ can create the list, but will have to manage it more efficient.