Distribution Technology Tradeoff Analysis

Rationalizing Document Messaging, Publish/Subscribe and RPC for Enterprise Distributed Computing

Under the wrappers of various distributed computing technologies there has emerged a clearer distinction between some basic paradigms and approaches that have, in fact, existed for some time. These are the RPC, Document Messaging and publish/subscribe paradigms. These distinctions are important because they present true architectural choices that can effect everything from the planning to deployment of an information system. The systems architect today has a set of choices and supporting technologies for each approach. Soap. WSDL and web services currently embrace both the RPC and document messaging paradigms.

Summary of the paradigms

Remote Procedure Call (RPC)

RPC has a history dating back to the early days of distributed computing and was brought into the object world with OMG-Corba, Microsoft-DCOM and more recently Java-EJB (Entity and session beans). The basic tenant of RPC is to make the distribution layer transparent to the programming language by making procedure calls flow through some kind of technology layer that handles distribution. All of these RPC mechanisms are object oriented (there are non object alternatives as well, but no longer mainstream) and can map directly to language interfaces. Soap-RPC is the web-services version of RPC based on Soap and XML.

Since RPC is based on the method call, RPC is inherently synchronous (There are some asynchronous ways to use Corba – but this is rarely used) and is designed to handle the data types typically found in programming languages. A remote interface with methods and method arguments is the basis for defining interoperability. Since most modern RPCs are object oriented, this is also known as distributed objects.

Soap-RPC differs from classic distributed objects in that there are no object references or ways to create distributed objects in Soap – all the "distributed objects" are assumed to be pre-defined. Corba and DCOM allow objects to be created remotely.

The Corba and DCOM flavors of RPC also support distributed transactions for tight coordination of an application with shared resources.

Think of RPC as like making a phone call to an automated attendant.

Document Messaging

The document messaging paradigm assumes a "document" as the basic unit of interaction, perhaps with attachments. Documents are sent between distributed systems, frequently to a static logical address. Document messaging may be connected or based on message queues for later delivery. Connected messaging may also be synchronous or asynchronous. Document messaging frequently assumes that the conversation between parties is long-lived and persistent (E.G. A business conversation).

Document messaging relies on some way to specify the documents used for interaction and XML has assumed premier status in this space. Messaging specifications (OMG-ECA, ebXML and WSDL) also provide a way to specify what documents may be accepted at specific service locations, essentially the interface of the messaging service. Various mechanisms have also been proposed do specify document ordering – such as OMG-ECA, ebXML-BPSS and WSFL.

With messaging there is no knowledge of methods or objects on the "other side" of a conversation, the binding between the parties is limited to the documents exchanged. It is expected that the "other side" knows what "method" or "application" to call for a particular type of document – or will reject it.

Think of document messaging as being like a web page (in the synchronous case) or like email in the asynchronous case.

Publish/Subscribe

Both RPC and messaging assume a conversation between specific parties, which is sometimes required (When you buy something you want to know the supplier!). However it is frequently unnecessary to have such directed conversations. Where there is just the need for one system (or person) to react to "events" within the enterprise (or within another enterprise) then publish-subscribe is used. Publish/ subscribe is similar to document messaging, but an intermediate messaging broker is used. Notice of interesting events are "published" to this broker and anyone with sufficient rights can "subscribe" to any event notification. The producers and consumers of event notifications do not even need to know about each other, they just need to know about the event and the service broker.

Publish/Subscribe is always asynchronous and frequently built on a messaging system backbone. Product names in this space are: MQ-Series, JMS, MSMQ and Sonic-MQ.

Think of pub/sub as being like a list server.

Criteria for picking the paradigm

Loose Coupling

Loose coupling means that the systems communicating can be implemented, evolve and be managed independently with minimal *required* common infrastructure. Loose coupling is absolutely required between independent business partners and between systems of different management domains. Unnecessary coupling between systems makes them hard to implement and very hard to evolve and change over time (and everything changes over time). Much of the problems with legacy systems has come from tight coupling, so loose coupling has been recognized as a good architectural principle even within one application – because even one application needs to have "parts" that are build and evolved independently.

High Cohesion

High Cohesion is the degree to which independent systems work together to further business goals – the "contract" between systems working together. The higher the cohesion the less chance there will be for inconsistencies to develop or for the underlying process to break because the supporting systems are not sufficiently aligned.

So cohesion, which is a business goal is a necessary and good thing, while tight coupling of the technology should be avoided.

Performance

Another important factor is performance. It should be noted that the performance difference between these paradigms is almost entirely subject to the supporting technology and the required resilience of the communication path. For example, A highly secure and reliable RPC system will be slower than a "best efforts" pub/sub. It is popularly held that RPC is "faster" than the other solutions, but this has not proved to be a reliable indicator. Multiple communications, buffering messages to disk, resource locking, security and guaranteed delivery are the costly coin of distributed computing.

In a large scale environment the effective use of resources must also be considered. Synchronous solutions will have faster turn-around time for a single message but will also place more demand on the infrastructure and may not scale as well – the system as a whole may be slower. Asynchronous solutions must be very well engineered to perform in an interactive systems environment – such as a user interface. Synchronous solutions must be very well engineered to scale to many users.

All distributed technologies have substantial overhead. Just the cost of "marshalling" data into some form of middleware protocol (be it XML or Corba-IIOP) and sending that data over a network to another process

is quite high and unavoidable. Each time a message is sent there is an overhead as well as each time a new connection must be made. Added messaging infrastructure like a disk-based queue for guarantied delivery and security adds to the overhead.

Due to the overhead of each message, distributed systems should use "large grain interactions", putting as much information in each message as possible, even if some of that information is not needed every time.

Where a real-time response is required a synchronous protocol may be preferred since it can send one document or method and get back a response in one connection. However, the additional system load of all those connections sometimes makes synchronous systems impractical for large numbers of users. This is an area of some debate.

Granularity

As discussed under performance – large granularity (big messages) is a substantial indicator of performance. It is also a substantial indicator of loose coupling. When interactions are large and have mechanisms for flexibility in their content, they are less coupled than a larger number of small interactions – since each interaction is a coupling point. So large grain interactions – sending as much data at one time as you can, is strongly preferred. Older EDI solutions were large grain, but lacked the flexibility of XML today so they become coupled. Solutions need some easy way to evolve messages without breaking existing systems.

Distributed Transactions

From a design point of view it would be nice if we could make a large distributed system behave like a local system, but this is not always possible. In particular distributed transactions have proved expensive to support and difficult to implement. It has become accepted best practice to make distributed interactions as discrete and atomic technical transactions wherever possible. Compensating transactions are used to maintain a consistent system state when "bad things happen". Within a single application distributed transactions may be unavoidable and will require supporting infrastructure.

Blocking

Blocking is the practice of having one resource wait while another does it's work. If that other resource is a human, blocking can consume a lot of resources. But even in the case of system-system interaction, a non-blocking solution will be faster from an overall systems perspective even if a single message may take longer. Blocking is caused by the use of synchronous interfaces and single threaded aplications.

Ease of development and maintenance

Ease of development is not an absolute – but depends on what you are starting with and what you are trying to accomplish. If you are starting with an existing object interface (that is sufficiently granular), RPC may be easiest. If you are starting with an EDI system or manual document system a document orientation may be easiest. If you are integrating legacy pub/sub may be the shortest course.

Besides the ease of development the total life-cycle should be considered, like cycle concerns will be addressed by loose coupling.

From a top-down business perspective the document centric messaging and pub/sub solutions seem to closely mirror the typical business model. The process of designing good documents for systems to interact makes sense to business people and can even be implemented without a computer system.

In programming the implementation an RPC system looks like method calls, so the knowledge transfer is very high. Document orientation can use generic toolkits (Like DOM or SAX) to parse and manage documents but these APIs can be clumsy. You can also convert documents into language objects with tools provided with by most systems. So if good tools are not provided, document oriented solutions may be a little harder to program. With good tools document oriented solutions are frequently faster to develop.

Best Practices Mindset

There is a subtle human difference between RPC and document messaging. Great architects of RPC systems have been making flexible and large grain interfaces for years, and these can perform and evolve very well. However, the mindset promoted by an RPC is that of a method – which tends to be small and very fine grain – primitive data types as parameters to lots of methods. It is very hard to change these interfaces over time – doing so breaks both sides. On the other hand – document messaging focuses the attention on creating a few large robust documents. XML provides almost "automatic" extension capability in the document paradigm.

So while you can define flexible large grain interactions with an RPC and you can make fine-grain documents – the XML messaging and pub/sub paradigms encourage and support best practices better than RPC.

Making a choice

Pub/Sub

Where there are systems that may have "side effects" from other systems (or business units with side effects from other business units) and there is not a requirement for a long-lived conversation – publish/subscribe is the bast choice in that it is very loosely coupled and performs well since there is no "blocking" of synchronous calls. The design of these systems is then centered on the event types and subscription rules. Since Pub/Sub does require a shared pub/sub domain, it may not be appropriate for B2B interactions – it works very well inside a single enterprise. However, if an interaction is or may be B2B one of the other choices may be better.

Note: It is possible to use pub/sub between independent business partners with a close working relationship. Sometimes automated EMAIL is used for this purpose.

Document Messaging

Where systems may be interdependent – requiring either a request/reply or long lived conversations, messaging may be the best choice. Since messaging may be either synchronous or asynchronous, the same application can be built to use either communication style and configured for best performance.

Document messaging works well in support of B2B, EAI, EDI and user interfaces interacting with back-end services.

Use of XML is very effective for messaging (and pub/sub) since it provides mechanisms for document extension, an application need only be concerned with a portion of a document and may ignore other parts – allowing the document and the end systems to evolve more independently. The process of designing effective documents aids in the understanding and loose coupling of the entire system. Since XML can be used for both messaging and pub/sub, the design and support infrastructure can be shared between the two.

RPC

If you are starting with DCOM, Corba or EJB (Entity or session bean) you already have an RPC and converting it to Soap may be automatic.

If you are starting with an existing object or legacy interface it may be most effective to directly expose that interface using RPC. There are good tools for making object interfaces into remote services. However, be aware that an object interface designed for local interaction may not work well as a distributed interface – they should still be large grain unless the systems are all on one small LAN.

If you are building an application and want to stay within the paradigm of that language, RPC will be easier to develop as it does not require designing documents.

In general we do not recommend use of RPC unless legacy concerns make it mandatory. The tighter coupling of RPC systems will almost always be a hindrance as the system evolves. Synchronous document

messaging is just as fast as RPC and much less coupled. When faced with an RPC or API interface, it is best to combine multiple method calls together to produce a large grain document.

Soap Messaging and RPC

The Soap protocol supports both messaging and RPC. The basic soap header provides for messaging – getting a document from one place to another. Layered on top of this is a way to represent programming language data types in a document instance and a way to bind to methods with signatures. So Soap RPC is basically two Soap messages used and combined in a particular way.

Note that when using Soap-RPC the "object" you get out of one side will have the same data as the object on the other side, but may have a very different interface – so care must be taken to not assume to much about the interface in the implementation.

Soap RPC is designed to be driven from a programming language interface and basically makes a document for each method, with the method arguments being elements of that document. In Soap RPC you do not design XML, you design object interfaces that are mapped to XML. You do not use Soap-RPC to send "generic" XML documents – you use it to export object interfaces.

Since Soap-RPC uses tools to make soap massages it can't be faster than messages as some people believe. Soap messaging and Soap RPC over the same transport will perform about the same. You may want to use Soap messaging over a synchronous protocol (like HTTP) if you want fast turn-around. Soap messaging over a queuing infrastructure will have a higher overhead than HTTP.

If you would prefer to use typed "language objects" to interface with XML rather than generic interfaces there is some help. In Java you can use "JAX-B" to convert a document to Java objects. In Microsoft you can use BizTalk for the same purpose. In both cases you will get language objects based on a DTD or XML-Schema input. One call converts the XML into these language objects.

These factors, combined with the best practices "mindset" make soap messaging our first choice. Evolving standards will also provide for Pub/Sub over soap messaging.

Summary of paradigm features

	RPC	Document Messaging	Publish/Subscribe
Synchronous?	Almost Always	Sometimes	Never
Sender and recipient connected?	Yes	Yes – perhaps transiently	No
Long-lived conversations	Not Usually	Frequently	Sometimes
Distributed Transactions	Frequently	Not Usually	Never
Turn-around time	Fast	Synchronous – Fast Asynchronous - Medium	Slower due to intermediary – usually disk buffered
Coupling	High – data types and methods must match between parties	Low – requires known party to message to	Very Low – requires only shared pub/sub domain
Specification	Object Interface	Document & flow	Event type
Required Infrastructure	Object Broker or Soap-RPC infrastructure	Communications path	Pub/Sub engine

Recommended for	Legacy	B2B, collaboration and User Interface	Collaboration & Integration