1.1 Transactional Integration

It all starts with a business requirement – as it always should. We have an ordering system (say the same one from the Knot anti-pattern) and the business says they only want to confirm an order to the user if the item is already secured for that order in the stock. From the technical point of view we have 2 separate services - one handles orders the other handles the stock – now what?

Figure 10.1 A vanilla ordering scenario. An Ordering service needs to confirm item in stock before confirming order for customer.

This sounds like a text book case for using transactions but in reality it isn't. I am going to explain why in a short while but before we go there let's do a (very) short recap on transactions and distributed transactions.

Transactions basically build on four basic tenets: Atomicity, Consistency, Isolation and Durability (ACID):

- Atomic – "All or nothing" meaning that once a transaction ends the state is either completely done (commit) or undone (abort)
- Consistent – The actions included in the transaction are done together so the state is kept consistent. If you were to remove something from stock and add it to shipment in the same transaction you won't have a situation where the item was removed from stock and not added to shipping list
- Isolated – while the transaction is in progress, logic which is not with part of the transaction will not see the world in its inconsistent form
- Durable – The consequences of transaction are saved to persistent storage so that they are available after a system restart

The simplest way for transactions is what is known as pessimistic locking. In this case a writer can only write if no one else is reading or writing and a reader can only read if no one else is writing (for a specific piece or block of data). In order to ensure ACIDness you need to write the data twice; once where you want it to end up and a second to a log file. This double bookkeeping ensures that if a crash occurs before the transaction was finalized (committed or aborted) you can check to see that both copies match and if not either (re)apply the log or rollback the data

Unfortunately pessimistic locks rarely work in real life scenarios so more advanced ways of locking and still maintaining ACIDness were developed. However, all the mechanisms holds resources for the transactions and all locking mechanism build on the assumption that the time spent inside the transaction is short.

The plot thickens further, when it comes to distributed transactions. Now we have at least two transactional resources and not only do each of them has to handle the transaction we also need to coordinate the state between them – since if one commits the transaction and the other rolls it back the overall transaction is incomplete. Still computer scientist where smart enough to come up with several

solutions to achieving distributed consensus and gave us two-phase commits, three-phase commits, paxos commits and whatnot. Case closed we can use transactions in SOA and life is beautiful.

Or is it?

1.1.1Consequences

Well first off the transactions, even the distributed ones are not a problem in themselves. For instance chapter 2 introduced the transactional service pattern to allow handling incoming messages in a reliable manner. The problems begin when the transaction scope involve more than one service or in other words:

> **Transactional Integration is an Anti-pattern where transactions extend across services boundaries (i.e. not isolated inside services)**

So what sorts of problems can transactional integration introduce to your SOA? Quite a few actually, with the main three being Performance problems, Security threats and rigidity let's take a look at them one by one.

With all the goodness transactions does it also introduce temporal coupling i.e. the need for all the involved actions to finalize on or about the same time. Even if the locks held while the transaction enfolds are permissive (optimistic), the coordination that is needed to ensure consistency needs to be synchronized. When you develop a non-SOA you may be able to take all the performance considerations at design time and make sure the system behaves. I'd still say distributed transactions are not highly recommended even then, since the rigidity of the consistency needed when trying to achieve a distributed consensus can still mean holding locks for a long time in cases of partial failures.

The situation is much worse in an SOA since each service can and will evolve independently both in terms of deployment and functionality. For instance, what will happen when the stock service moves to another datacenter (e.g. ported to the cloud). What if the designers of the stock service decide that when the stock level hits a thresh-hold they want to automatically order new supplies and they want that in a transaction – Now you cannot secure an item in the stock until new supplies are ordered. All of a sudden our transaction expanded and now includes the ordering service, the stock service and a supplier' service(s). So one risk is that designers of services participating in our transaction extend the transaction to handle business rules they need to comply with.

Another risk highlighted by the above mentioned scenario is related to security. In the example we added the suppliers' services into our transaction. This means that we now run the risk that external systems will now hold locks on our system, either maliciously or by neglect, effectively creating a denial of service scenario on our services. Service boundary, its edge, should also be a trust boundary, externalizing transactions to third parties might be far-fetched but externalizing it to other teams within the organization which work on their own services with their own priorities is not, the same risk applies there.

The last risk we can highlight with this example is connected directly to the Knot anti-pattern (which is one reason the same sample scenario is used) – Having transactions between services increases the coupling between them and increased coupling increases the risk of ending up with a Knot, which effectively kills SOA.

One can argue that most of even all of these are hypothetical situations and that when we design the SOA we can take the real constraints into consideration and plan for them – isn't that why we have enterprise architects for? Though the scenarios are over-simplified to illustrate the problems in a clear-cut

way, real life scenarios manifest the same problem in subtler ways. The main point is that evolvability and flexibility are the hallmark of SOA – That's why we want an SOA in the first place - so that we can evolve the IT of the organization to better match the **changing** needs of the business. The end result is that regardless of how we plan it out on the onset, in the long term it is hard to control who participate in the transactions which means that adding distributed transactions to the mix is an accident waiting to happen.

1.1.2Causes

The main reason Transactional Integration happens was already mentioned above, when we start out and design our SOA we have a relatively good grasp on the enterprise business. Again, the problem is that an SOA solution is not static – if it is than it is probably a needless overhead to architect it as an SOA anyway.

Even if you do have a good initial understanding of the business flows, that understanding can deteriorate pretty fast. It isn't just that requirements change over time – an even greater force of change is getting deeper understanding as to what is exactly needed. To side track a little, the pragmatic way to implement an enterprise-wide SOA is not to do a multi-month (if not multi-year) project just to documents and design the overall architecture and services and only then begin the transition to the new architecture. Doing that is like sending a message that new things should just sit there and wait until you'd be ready – I am yet to see a business that can afford that. No, the more realistic and cost-effective way is to do some upfront design but also begin developing real services and work them into the existing software portfolio. This is sort of like building a new intersection where you also have to build detours, keep some of the lanes open - anything to keep the traffic going. When you work on an SOA in this manner the rework, the growing understanding of the business and the requirements changes means you can expect a lot of evolution to happen, as mentioned above, transactional integration will make that unlikely or very hard.

Other forces pushing to the transactional integration anti-pattern are the marketing organization of technology vendors. That's an odd statement so let me explain. Technology vendors provide, well, er technology. Thus when a vendor ships a technology frameworks, it is usually geared to answer a broad variety of needs. Take for example, Microsoft's Windows Communication Foundation (WCF) which is a unified infrastructure for remote communications between components. WCF offers message based communications along support for names-pipes; it is built to replace RPC technologies like remoting and it supports SOAP (WS*) web services, some support for REST style services and what not – yet WCF is by and large is marketed as an "SOA foundation". This isn't to say you can't use WCF for SOA but it does a lot more, it also does transactions… Other vendors follow the same path, whenever there's a new buzzword, their marketing organizations take whatever technology they currently have and slap that on it. The end result of that is a lot of confusion in regards to what right and what is not. The use of transactions for cross service integration is, unfortunately, just one sample of this effect.

Well, if transactions are not the way to go, what can we do instead?

1.1.3Refactoring

There are several options to get around the problem either using Orchestration or Sagas, Inversion Communications pattern and similar means to achieve eventual consistency.

But wait, what exactly is the problem we're trying to solve? We are trying to achieve distributed consensus and consistency in the data and business picture as seen by several services. Let's take a quick look at the business scenario presented above. We have an ordering system and the business says they only want to confirm an order to the user if the item is already secured for that order in the stock.

One way to solve this would be to externalize both the transaction scope and the business flow to an orchestration engine (see Orchestration pattern in chapter 6). The advantage over transactions directed from within the services is that the orchestration engine has the full picture both of the services involved (and their various trust levels) and the flow of which service calls to which service so there's more control on who does what and when. Still services participating need to be transaction aware and need to retain internal locks for external constraints so use this with caution.

Another alternative is to use sagas (see Saga pattern in chapter 5), Sagas basically means running a long running interaction (where messages are related and belong to the same "conversation) but without holding the same transactional guarantees as ACID transactions. For instance in case of a stock problem the ordering service will have to do a compensating action to handle the error. In order for this to work in a reasonable manner the services may need to hold some data about the world such as a some data about stock levels so it can receive a reasonable decision on its own.

Sagas can be augmented by the inversion communications pattern to make the services send events of their actions and other events to subscribe to them to create choreography scenarios. In our example the order service would publish that it has a new order that needs handling and the stock service would listen on that. Once the stock secures the items it will publish an event that says that so the Ordering service can notify the customer that the order is ready (maybe there would be additional steps like actually shipping the product etc.)

Both Sagas and the Inversion Communication pattern actually implement an eventually consistent system – we basically relax the temporal constraints on decision making by the various services. This can, and usually does, translate into how the business works in general. In our ordering example it may mean that it would be better to send an additional notification to the customer that her order was received, when the order service processed the order and sent the event to that affect

1.1.4Known Exceptions

I can't think of a lot of SOA solutions that would benefit from cross service transactions. Transactional integration is usually a bad idea for most distributed systems anyway – from similar reasons to the ones mentioned above.

A rare exception to this rule might be for a closed solution (i.e. a system and not an organization) that is building on SOA principles. In a closed environment where everything is controlled it might be possible to pull it off without suffering from the rigidity and performance problems induced by Transactional Integration. Even in these rare cases it would still be preferable to control the transaction scope outside of the service by using an orchestration engine. Using orchestration means that at least the scope of transactions and the general flow of the business processes will be handled in the same place.

I would still be wary of going down this path since even closed control system tend to evolve over time so be forewarned.

A related anti-pattern, which bears some resemblance to Transactional Integration, is the Whitebox Services anti-pattern. An anti-pattern which occurs when services expose their innards too much