5.4 Saga

Back in chapter 2, we talked about the Transactional Service pattern as a way to help make a service handle requests in a reliable manner. However, using the Transactional Service Pattern, only solves one part of puzzle. Let's take another look at the scenario that was presented in chapter 2. Figure 5.8 below shows an Ordering service that processes an order. The interesting issue here comes from steps 2.3 and 2.4. Within the internal transaction of handling the request, the Ordering service has to interact with two other services: request a bill from an internal billing service order stuff (e.g. parts or materials) from an external supplier.

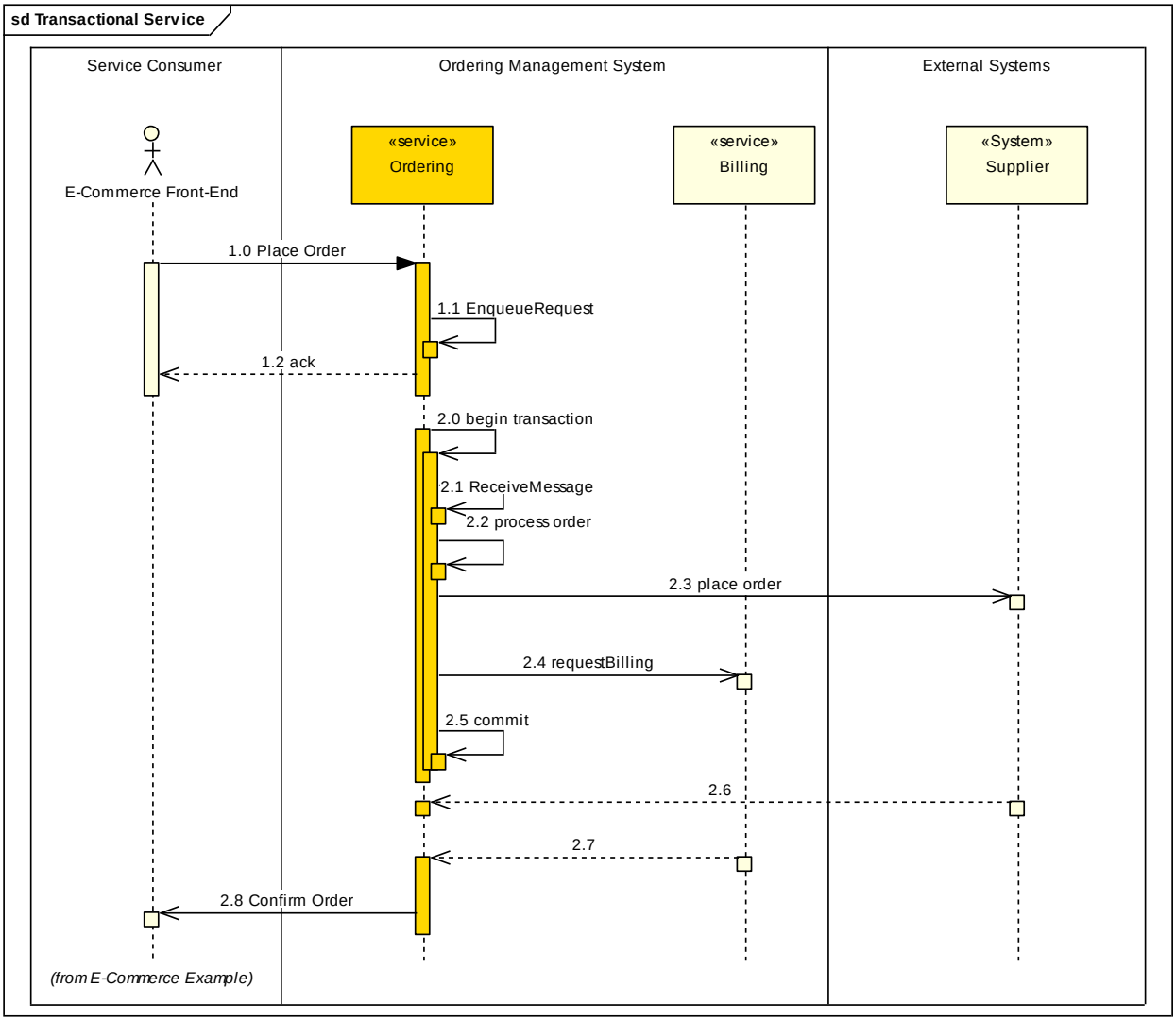

Figure 5.8 Sample message flow in an E-Commerce scenario (talking to an Ordering Service). The front-end sends an order to an ordering service which then orders the part from a supplier and asks a billing service to produce bill the customer. Note that all the actual handling of the Place Order message is done within a single local transaction.

Fine, so what's the problem? There are 2 major problems lurking here. One is what will happen if instead of committing the internal transaction at step 2.5 the Ordering service decides to abort its (internal) transaction?. The other problem has to do with getting some commitment from the other services so that the ordering service can continue its work based on that commitment. For instance we may want to get a confirmation from the supplier that she secured the items we ordered for us before we confirm the order for the customer

5.4.1 The Problem

The obvious answer to the two problems mentioned above is to extend or flow the internal transaction which the ordering service already has into the other services. This "extended transaction" is known as a "distributed transaction". Using distributed transactions, the ordering service would have call both the billing service and the supplier's system as part of a single transaction and if all the services agree to commit the whole transaction is committed and completed together. This sounds really, really great (really ☺), we even have the technology to do that, which, by the way, predated SOA by many years.

But, and there's always a but… , what if the supplier can only complete their part of the transaction after a senior manager will authorize the deal ? Can we hold all our internal locks waiting for that manager to return from his vacation in the Bahamas sometime next week? Probably not. Even more so if this supplier is also a competitor. Now, they might prolong the transactions just to put us out of business since we hold locks on our internal resources while we wait for them to complete the transaction. The specific scenario I painted might be too farfetched but the point is that we can't make assumptions on how other services operate. This is especially true for services we don't own. There are additional reasons not to do cross-service transactions and you can read about them in detail in the Cross-Service Transactions anti-pattern (in chapter 10).

Even if you think that cross-service transactions are not problematic as a concept. You would probably agree that long transactions are not very good. So the more conversational the interaction between the services gets the more we need to think about alternatives atomic transactions. Again, if we look at the scenario in figure 5.8 we currently have 2 messages going out from the ordering service – which might be borderline in terms of number of interactions. However business processes can sometimes involve much more elaborate conversations.

A lot of messages flowing might be the sign of the a Chit-Chat anti-pattern (chapter 9). Nevertheless, few and sparse interactions are not realistic either. Services rarely live in complete isolation, after all, as mentioned in chapter 1, interoperability is one of the reasons we went with SOA in the first place. This mean we need to have a way to handle complex service interactions in a reliable way – without bundling the whole thing in one lengthy atomic transaction.

To sum both the problems we've see thus far, what we want to know is :

> **How can we get transaction-like behavior or complex interactions between services without transactions**

I think by now it is clear that using a single transaction is not an option. If all the services involved are under your control you might want break the long process into multiple steps and run each step in its own transaction. Smaller distributed transactions are definitely a step in the right direction. But we are still bound by cross-service transaction –and , because everything is not bounded by one single

transaction we have problems like canceling the effect of a first step if something failed in the third or fourth one.

Another option is to try to model our contract so that we will never need this kind of complex interactions. The way I see it though, is that we can minimize interactions by increasing the granularity of the services – however, if there's also a limit to how large we want our service to be – we don't want to end up with a single monolith service that does everything. In my experience, Services

The option we are left with is to break the service interaction, that is our business process, to a set of smaller steps and model that into a long running conversation between the services.

5.4.2 The Solution

The Saga interaction pattern is about providing the semantics and components to support the long running conversation mentioned at the end of the previous section.

> **Implement the Saga pattern and break the services interaction i.e. the business process, to multiple smaller related business actions and counteractions. Coordinate the conversation and manage it based on messages and timeouts.**

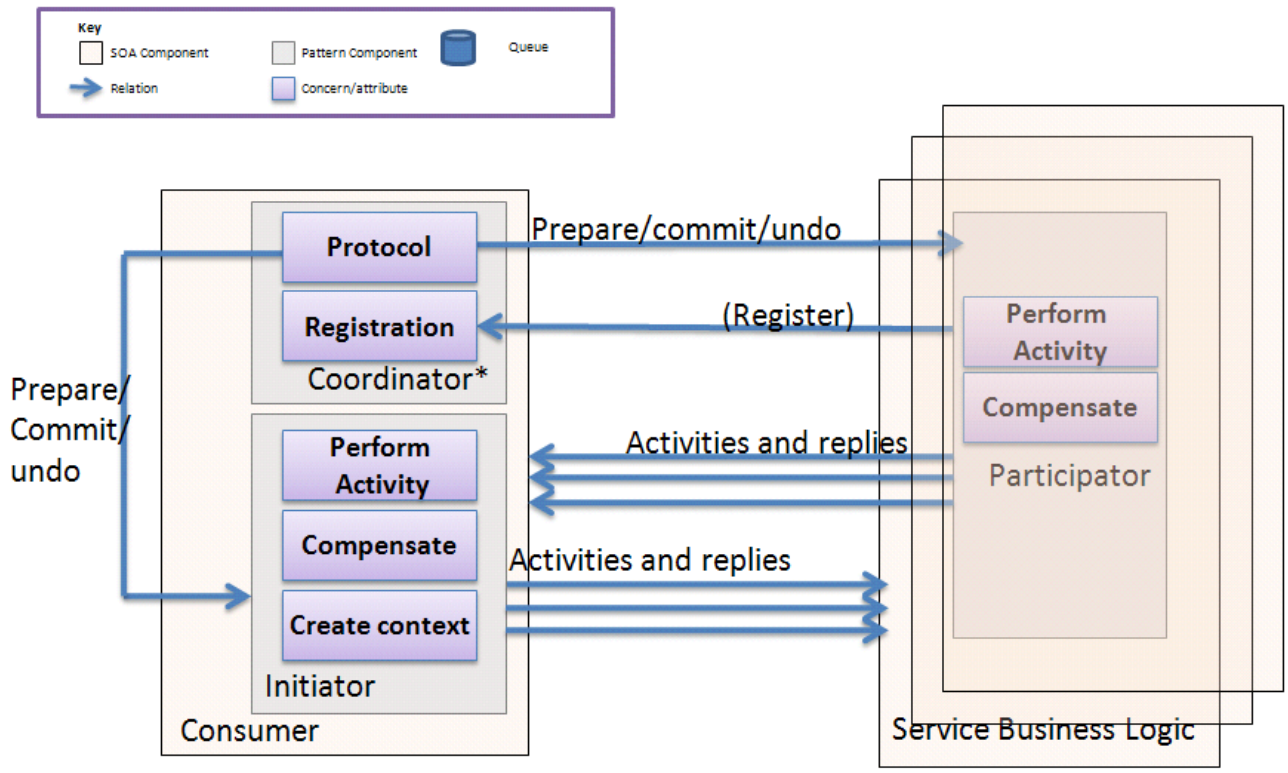

Figure 5.9 The Saga Pattern. A service consumer and one or more services hold a long running conversation within a single context (a Saga). Once the parties reach some consensus the conversation is committed. If there are problems during the conversation and the interaction is aborted. The involved parties perform corrective steps (compensations)

Hector Garcia-Molina and Kenneth Salem defined the term Saga back in 1987 as a way to solve the problem of long lived database transactions. Hector and Kenneth described a Saga as a sequence of related small transactions. In a Saga the coordinator (database in their case) makes sure that all of the

involved transactions are successfully completed. Otherwise, if the transactions fails the coordinator runs compensating transactions to amend the partial execution. What made sense for databases makes even more sense for service interactions in SOA. We can break a long service interaction into individual actions or activities and compensations (in case of faults and errors)

The first component we notice is the initiator. The initiator triggers the Saga pattern by creating the context, the reason for the interaction. It then asks one or more other services (participators) to perform some businesses activities. The participators can register for coordination (depending how formal the Saga implementation is). The participants and initiator exchange messages and requests until they are reach some agreement or they are ready to complete the interaction. This is when the coordinator requests all the participant (including the initiator) to finalize the agreement (prepare) and commit.

If there was a problem either in during the interaction or the final phase the activities that occurred have to be undone. In regular ACID transactions you can rollback – however in a Saga you have to perform a counteraction, called compensation, which contrary to Newton's law[1], may not be the exact opposite of the activity that has to be undone. For instance if the result of the original activity the service crossed some threshold it may not wish to undo the action it took. Another example is that cancelling the action may require something form the service(s) that requested the action in the first place (e.g. cancellation fee) or that too much time has passed which makes it impossible to undo the effect. If we try to look at an example from the real world- if a result of a Saga was to launch the space shuttle, a compensation would be to abort the mission and return the shuttle home – but you can't just pull it back into the launch pad.

The Saga pattern is sometimes referred to as "Long Running Transaction". It is true that you can conceptually think of a Saga as a single logical unit of work and that it does make use of transaction semantics. However a Saga doesn't really adhere to the transaction tenets like atomicity or isolation – mostly because the interaction is distributed both in time and in space. For instance when you call a compensation it might be too late to undo the original action so that either or it might have consequences like cancellations fees or partial deliveries. I think the term Saga better reflects the fact the interaction is lengthy and that the messages are related.

Let's take a look at a how the interaction of the ordering scenario we presented in figure 5.8 above might look like when we utilize the Saga interaction pattern. Diagram 5.10 below demonstrate a scenario where the supplier is out of stock for the ordered items. In this case both the ordering and billing need to be canceled. We also need to notify the front-end that there was a problem and to let the supplier know we closed the interaction.

Using the Saga pattern, all the services involved (Ordering, Billing and the Supplier's service) notify their state. For instance the supplier sends a fault message to let the ordering service know it had a problem processing its request. When the coordinator component inside the Ordering service gets the fault message it requests the other parties, i.e. the ordering service itself and the billing service to compensate and once that done it notifies the supplier that the interaction completed handling the fault.

Also note that notifying the front-end about the failure is done during the compensation of the ordering service. It is not a task of the coordinator.

[1] Newton's 3rd law of motion: For every action there is a equal and opposite reaction

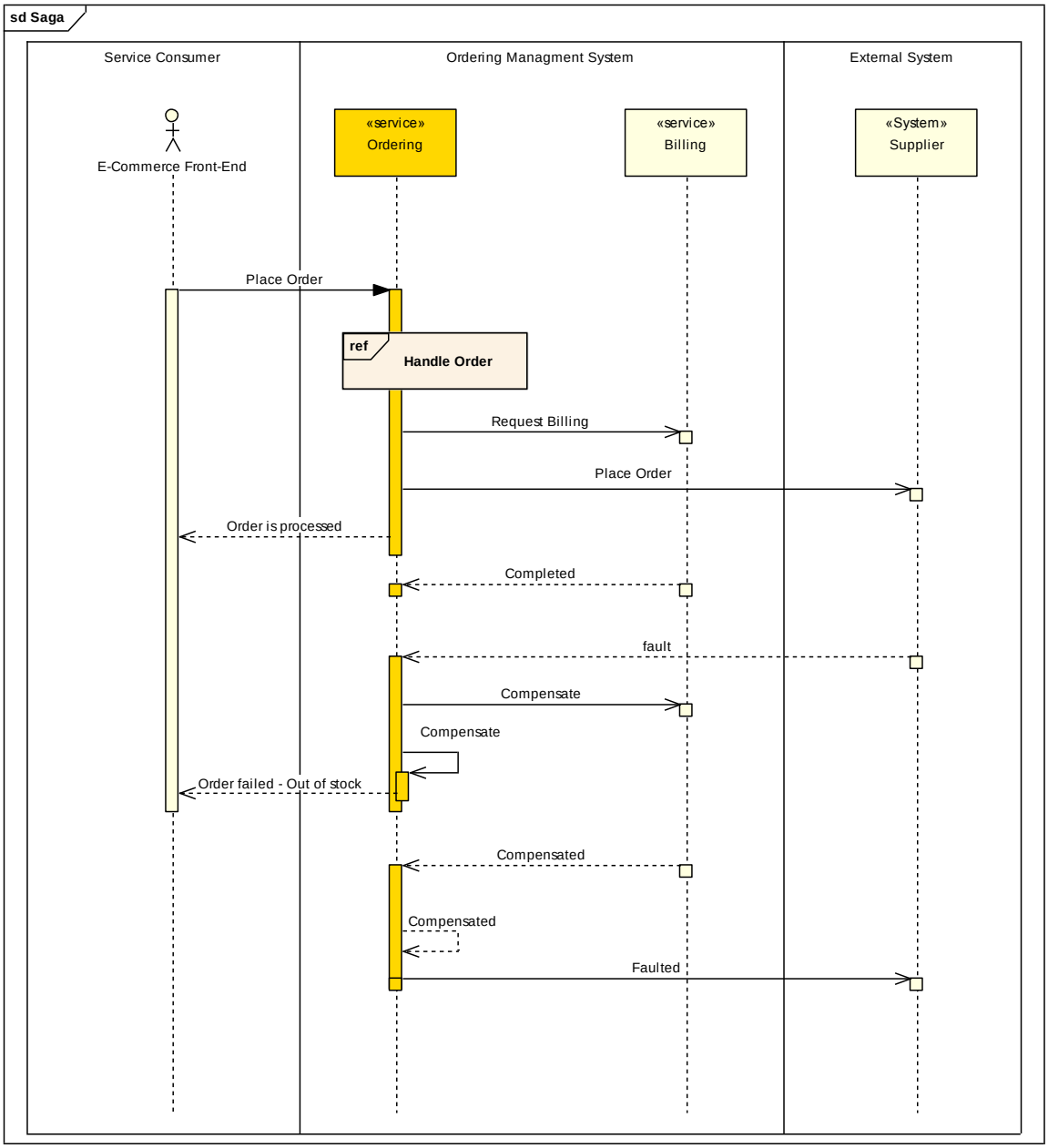

Figure 5.10 E-Commerce scenario from figure 5.X remodeled using the Saga pattern. The interaction with the billing and the supplier is now coordinated in a saga. And the ordering service can handle problems in a more robust way by canceling the order and notifying the front-end instead of hoping for the best

The interaction above has the service consumer and services control the interaction internally. One good option to do this is to utilize the Workflodize pattern (see chapter 2) so that each service holds a workflow internally which follows the sequence and different paths of the interaction. Other patterns related to the Saga pattern are correlated messaged (section 5.3) and Reservation pattern (section 5.5)

Another approach you can take to implement the Saga pattern using an external coordinator for the conversation– see the Orchestrated Choreography pattern in chapter 7 for more details. The semantic difference between an internally coordinated Saga and an externally coordinated one is that the services

involved in the first try to come to a mutual consensus while the services in the latter are driven to created a larger whole.

The main effort behind the Saga pattern is to decide on the business activities and compensations. You can use techniques such as Business Process Modeling (BPM) to form a good basis of what these activities might be.

Even though the main effort in implementing the Saga pattern is the business side, i.e. modeling business processes and activity that would support long running conversation, there are also a few technological aspects that have to do with the messages and protocol – let's take a look at them.

5.4.3 Technology Mapping

At the minimum the Saga pattern requires you to add compensation messages to any state altering message that can participate in a Saga. Again, it is important to emphasize that the compensation may not be able to undo the original activity – but it does have to try to minimize the effects of the activity. The innards of the processing of the compensation messages varies depending on what needs to be done to cancel the effect of the original message. Note that it is usually better to set statuses to cancelled rather than delete, especially at the database level, since the original action might have triggered other business processes and actions. For instance, if as a result of a message you added an order, another service might have produced a bill. Chances are that billing also occurred within the same Saga, but you might not know or control that within the ordering service. Making a change that leaves traces behind it (like cancel) is better than a delete since it also allows resolving problem manually if the need arises.

Another message type that is important for a Saga is failure message. When you have a simple point to point interaction between services the reply or reaction a called services sends is enough to convey the notion of a problem, the calling service consumer which understands the service's contract can understand that something is amiss and act accordingly. When you implement the Saga pattern however, you have the possibility of more than two parties and you also have a coordinator. The coordinator is not as business aware as the service's business logic but it does define control messages in order to understand the status of the interaction.

As you probably know (or at least notices by now) web-services are considered the primary technology for implementing SOAs and the Saga pattern is not different. The WS-* stack of protocol has produced the WS-BusinessActivity as part of WS-Coordination.

WS-BusinessActivity has two variants one which is a little more ordered and the other which is a little more loosely-coupled – with the cost being increased chances for compensation. The first is Business Agreement with Coordinator Completion – where the coordinator decides and notifies the participants when to complete and the Business Agreement with Participant Completion, where the participants decide when they complete their roles within the activity.

Unlike self-implemented Sagas, WS-BusinessActivity defines an orderly protocol and states for both the participating services and the coordinator. WS-BusinessActivity defines two coordination types – one, called AtomicOutcome where all the participants have to all close (comit) or compensate and another called MixedOutcome where the coordinator treats each participant separately. Additionally WS-BusinessActivity defines two protocols one where the coordinator decides when the participant fulfilled

their share of the business process and one which is more loosely coupled, where each participant decides when it has finished its part of the process. For instance, Figure 5.11 below shows the state transitions for a participating service using the WS=BusinessActivity with participant completion.

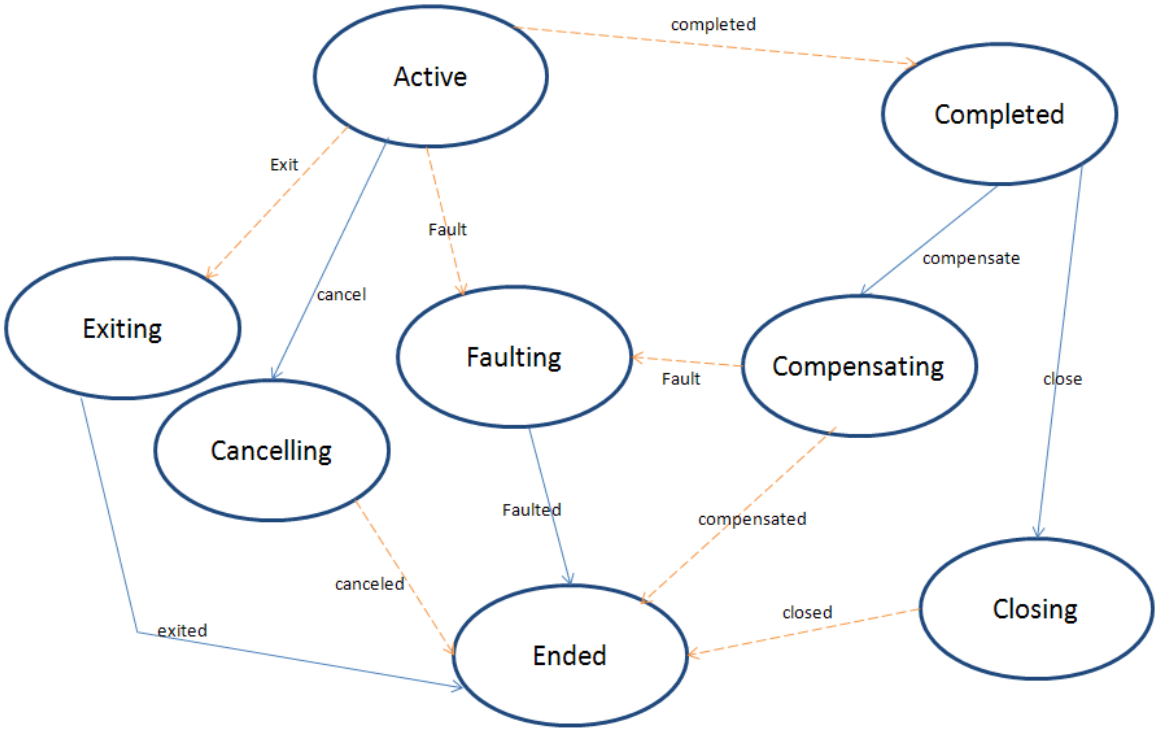

Figure 5.11 State diagram from the point of view of a participating service using the completion by participants variant of the WS-BusinessActivity protocol. The state transitions can be either the result of decisions by the service (the dotted lines) or by messages from the coordinator (the full lines)

Another important technology option for implementing the Saga pattern is to use BPEL (Business Process Execution Language) or it WS-* implementation known as WS-BPEL (or BPEL4WS in previous versions). Additionally you can also use a non-BPEL compliant orchestration engine. In any even these technology mapping fall under the external coordinator mentioned above and are covered in more depth as part of the Orchestrated choreography pattern in chapter 7.

5.4.4 Quality Attributes

The quality attribute scenarios section talks about the architectural benefits of utilizing patterns from the requirements perspective. The scenarios are used to describe the architectural requirements in a way that allows evaluating architecture to see if it answers them. Another use for the scenarios is to design an architecture – or from the perspective relevant here - as a way to identify situations where a pattern is applicable.

The main reason to employ the Saga pattern is to increase the integrity of the system. As I already mentioned in the sections above, transactions are problematic when it comes to distributed environment in general and even more so when using SOA. Nevertheless we still want to be able to coordinate the

behavior of services and get meaningful interaction. Be letting us coordinate the behavior and failure handling we introduce a reliable, predictable long-running conversations.

Another aspect of integrity which is a reason to use the Saga pattern is to increase the predictability. In a distributed environment it is relatively hard to know what the outcome will be, this is especially true if you use other patterns like Inversion of Communications (section 5.6). The Saga patterns allows introducing some control into the interaction and verify that the outcome of a complex interaction be along known paths (completed or compensated).

Lastly, the outcome of increased predictability is also increased correctness. Knowing how the system is going to behave it is easier to construct system tests to verify that the desired outcome indeed happens

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Integrity	Correctness	Under all conditions, an order processed by the system will be billed
Integrity	Predictability	Under normal conditions, the chances of a customer getting billed for a cancelled order shall be less than 5%
Reliability	Handling failure	Resuming from a communications disconnection , all the processes that were interrupted shall remain consistent

Table 5.5 Saga pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Decoupled Invocation pattern.

Writing compensation logic is relatively complicated as the timeline advances the number of changes in the service can get rather large, which makes it harder to get the predictability when you try to undo an early change. One way to try to cope with that is to implement the Reservation pattern, which is the pattern we are going to look at next.