Chapter 23
Towards Distributed Atomic Transactions over RESTful Services

Guy Pardon and Cesare Pautasso

Try-Cancel/Confirm: Transactions For the REST of Us

- Atomikos.com

Abstract There is considerable debate in the REST community whether or not transaction support is needed and possible. This chapter's contribution to this debate is threefold: we define a business case for transactions in REST based on the Try-Cancel/Confirm (TCC) pattern; we outline a very light-weight protocol that guarantees atomicity and recovery over distributed REST resources; and we discuss the inherent theoretical limitations of our approach. Our TCC for REST approach minimizes the assumptions made on the individual services that can be part of a transaction and does not require any extension to the HTTP protocol. A very simple but realistic example helps to illustrate the applicability of the approach.

Introduction

The Uniform Interface (Fielding 2000) of a RESTful Web service (Richardson and Ruby 2007) implemented using HTTP has very useful and positive implications on the reliability of the interaction of clients with a service following the constraint. Considering that `GET`, `PUT`, `DELETE` requests are by definition idempotent, any failure during these interactions can be addressed by simply repeating the request.

This property, however, cannot be directly applied in a service composition scenario (Pautasso 2009) where multiple interactions between a set of RESTful services need to happen atomically. Even if a single idempotent interaction between one client and one RESTful Web service is reliable, it is not clear how to guarantee the same property of atomicity when a client is interacting with multiple RESTful Web services. This problem is the central topic of this chapter, and will be discussed by means of the running example illustrated in the following section.

G. Pardon (✉)
ATOMIKOS, Hoveniersstraat 39/1, 2800 Mechelen, Belgium
e-mail: guy@atomikos.com

E. Wilde and C. Pautasso (eds.), *REST: From Research to Practice*,
DOI 10.1007/978-1-4419-8303-9_23,

Example: Booking Two Connecting Flights

Suppose we want to book a flight composed of two connecting flights from two different and autonomous airlines: swiss.com and easyjet.com, via some travel agency service acting as a service composition over the two airlines. Let's assume that both airlines have the same hypermedia contract for bookings (for reasons of simplicity, and without loss of generality since the composite service is supposed to know all of the hypermedia contracts involved). The REST implementation of the airline information and booking services could be designed as follows.

Checking Seat Availability

Clients can inquire about the availability of seats on a flight at the URI: `/flight/{flight-no}/seat/{seat-no}`. For example, the `GET/flight /LX101/seat/` request will return a hyperlink to the next available seat on the flight `LX101` or none (e.g., `204 No Content`) if the flight is fully booked.

Booking a Seat

A `POST` request to the `/booking` URL with a payload referencing such seat will create a new booking resource and redirect the client to it by sending a hyperlink identifying it such as `/booking/{id}/`. The body of the request can contain a reference to the chosen flight and seat (i.e., `<flight number="LX101" seat="33F"/>`). The booking can be updated with additional information using a `PUT/booking/{id}/` request.

Composition of Bookings

We are now ready to present the first user story, which will be our motivating example throughout this chapter.

Story 1 *As a customer, I want to book a composed flight consisting of two independent, connecting flights from both airlines.*

It is the responsibility of the travel agency composite service to satisfy this requirement. A straightforward implementation (without a transaction model for REST) would be the following:

1. `GET swiss.com/flight/LX101/seat/`
2. `POST swiss.com/booking`
3. `GET easyjet.com/flight/EZ222/seat/`
4. `POST easyjet.com/booking`

The problem is that it may happen that after the first airline service has successfully performed the booking (step 2), the second airline may reply that there are no seats available. Thus we have only a partial flight.

Even if we reorder the requests as follows:

1. `GET swiss.com/flight/LX101/seat/`
2. `GET easyjet.com/flight/EZ222/seat/`
3. `POST swiss.com/booking`
4. `POST easyjet.com/booking`

the problem is not solved. Even if both step 1 and 2 may return a link to an available seat, the following booking requests may fail due to concurrent intermediate bookings. Thus, we may still end up in a situation where we have reserved one flight but not the other one. If 3 fails (due to, say, intermediate bookings at easyjet.com between steps 2 and 4) then we have one flight but not the other one. The retry of individual requests does not help here: we can try to repeat step 4 as many times as we like, but if the flight is fully booked then we will keep getting the same failure each time. What we really need is the ability to make step 3 and 4 tentative, so that they can be confirmed later. This way the whole process becomes atomic and happens entirely or not at all.

Our Goal: Lightweight Transactions for REST

The goal of this chapter is to propose a solution to the problem of atomicity within distributed RESTful interactions within the constraints of: (a) Using a lightweight transaction model (Pardon and Alonso 2000) based on ATOMIKOS TCC (Pardon 2009); (b) Minimizing, or in the best case, avoiding changes to the REST uniform interface and the HTTP protocol. (c) Assigning to the service running the composition the responsibility of ensuring the atomicity of the transaction.

A solution should provide the ability to transparently group multiple RESTful interactions and treat them as a single logical step, as well as to ensure that the consistency of a set of resources which are distributed over multiple servers can be kept. Whereas solutions have been proposed to batch interactions affecting multiple resources provided by a single server [e.g., WebDAV's explicit locking methods (Goland et al. 1999), or the transactions as a resource approach from (Richardson and Ruby, 2007, p. 231)], these are not directly applicable to interact with multiple resources distributed across multiple servers.

About this Chapter

This chapter contribution focuses on addressing the atomicity property of distributed transactions across RESTful Web services. This already satisfies the requirements

of a wide class of applications, where atomicity is a necessity, while isolation is not. For example, all scenarios involving some kind of resource reservation are relevant, since once a resource is reserved within a transaction, its reserved state should become immediately visible to other clients in order to avoid overbooking. Our solution is thus applicable whenever clients need to atomically perform a purchase (or more in general, change the state) of a set of distributed and autonomous resources.

The rest of this chapter is structured as follows: in "A Transaction Model for REST" we use our running example to further define the business-driven case for REST transactions and then discuss the technical requirements that a solution should satisfy. In "Protocols" we outline the transaction protocol, which is discussed at length in "Discussion". Finally, we give a brief survey of related work before drawing some conclusions.

A Transaction Model for REST

Whether or not REST needs transactions has been heavily debated within the REST community (Little 2009). We claim there is a clear need, and we try to motivate it here. Our motivation is in two parts. First, we define a business model for RESTful services that needs transactions. Next, we define the technical qualities that we think a transaction model for REST should possess in order to be successful.

Why REST Needs Transactions

With the first story we have already motivated the need for atomicity, and why idempotent requests are not enough. We will now refine this model based on realistic business needs of each of the parties involved.

Refining our Example: Confirmation of Bookings

As hinted in the introduction, we need a way to make bookings tentative until confirmed:

Story 2 *As a customer, I want to be able to confirm a booking when I am done. Bookings that are not confirmed are not billed to my account.*

Confirmation can (and should) be business-specific. In the context of our running example, we assume that a confirmation hyperlink is returned by the RESTful API of the airline service (e.g., in response to a `GET/booking/{id}` the service returns `<flight number seat><payment uri="/payment/X"></flight>`). Thus, the booking can be confirmed with a `PUT/payment/X <VISA ...>` request.

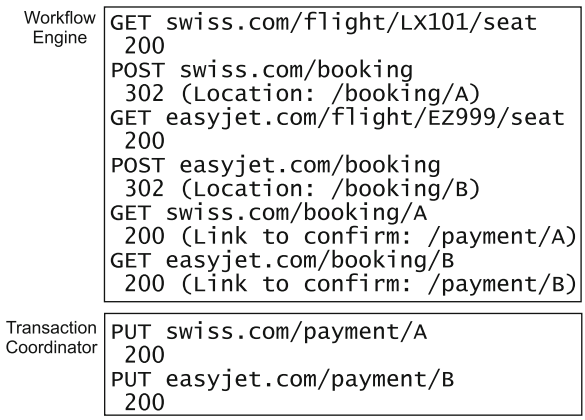

Fig. 23.1 Example of an atomic reservation for two flights (happy path)

Transactional Booking Workflow

The travel agency can now implement a transactional booking as shown by Fig. 23.1.

In terms of design, the first set of interactions can be driven by the workflow that composes the two services, while the final confirmations to close the transaction could be sent to a transaction coordinator component.

What if Step 4 Fails?

Let's return to the original problem: what if step 4 (i.e., the second booking) fails? By not performing any confirmation, the workflow engine ensures that no billing is done for either flight. This avoids our original problem as the transaction coordinator will not confirm any of the bookings.

Refining even more: Cancellation of Bookings

Confirmation is driven by the needs of the customer and the travel agency that composes the individual services. From the point of view of the airlines, an additional story arises:

Story 3 *As an airline, I do not want to wait for a confirmation forever. In other words, I want to be able to autonomously cancel a pending booking after some timeout expires.*

This should be obvious: as an airline, I do not want to loose money because some travel agency keeps seats reserved without confirmations. Consequently, we need a cancellation event triggered by some timeout specific to the airline.

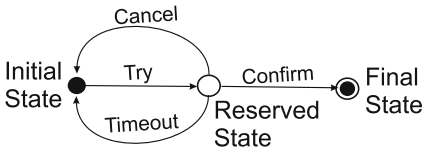

Fig. 23.2 Generic state machine of a resource complying with the Try-Cancel/Confirm protocol

The REST implementation could be as follows: `GET /booking/{id}` returns `<flight number seat><payment uri="/payment/X" deadline="24h"></flight>`). The composing workflow service can use the deadline as a hint to when the expiry of the reservation will happen.

Generalisation: Try-Cancel/Confirm

Our stories are particular illustrations of the more general Try-Cancel/Confirm (TCC) protocol. As shown in Fig. 23.2 each request is "tried" and remains tentative until it is either confirmed or cancelled. Composition of TCC services leads to a natural, loosely-coupled transaction model. Cancellation may occur spontaneously after a timeout or might be triggered by an external event (the latter we consider out-of-scope for this chapter).

Although originally formulated by Pardon (2009), a similar model seems to have been discovered independently at Amazon (Helland 2007) – which supports our vision about TCC's broad applicability and relevance.

Technical Requirements for REST Transactions

Industry practice has shown that transactions need to be non-invasive or they will be avoided. This is mostly due to the tight coupling and the additional complexity they introduce in the design and implementation of services which can participate in a transaction.

Our simple proposal attempts to avoid the negative impacts of existing approaches while ensuring that the visibility and the interoperability that have come to be expected of RESTful services are not affected.

Loose Coupling

The resources published by a RESTful Web service are typically seen as independent entities whose state changes happen autonomously from one another (Richardson and Ruby 2007). Clients interacting with resources may change their state trough atomic interactions which however do not span across multiple resources (Helland 2007).

The main constraint for our proposal is to ensure that resources remain autonomous and that performing transactions over them does not introduce any

additional coupling among them. This is important to remain within the scope of the REST constraints which emphasize the role of the client as the one driving forward the state of an application.

A transaction solution for REST is considered loosely-coupled (Pautasso and Wilde 2009) if participating services are unaware of the fact that they are being part of a global transaction. More precisely: the individual participating services do not need to have any additional knowledge or implement any extra protocols besides what they already support. Whereas not all RESTful services may be able to participate in such transactions, we claim that there is a significant number of resources that naturally fit with our assumptions due to the nature of the business service they implement. This is particularly true for services that comply with the TCC business model outlined in the previous section.

No Context Please

Avoiding to make use of an explicit transaction context is a radical departure from most distributed transaction protocols which assume that a transactional context needs to be established and maintained among the participants, which must be aware of the transaction and thus become tightly coupled with one another.

One of the most important requirements to ensure loose coupling is that there should be no transaction context shipped around, thereby eliminating a lot of shared state interpretation and hidden dependencies among services. Most existing protocols for distributed transactions rely on such mechanism to establish a context shared among the participants. Thus the services become aware of participating in a transaction and must carry the burden of maintaining such context. Our goal is to define a protocol which removes the need for establishing and maintaining such context.

Align with the Business Functionality

The classical ACID transaction paradigm revolves around database locks and low-level rollback at the database level (Bernstein et al. 1987; Gray and Reuter 1993). Distributed ACID transactions (i.e., involving more than one database backend and/or service) usually require a "distributed transaction coordinator" to drive the individual ACID transactions via the XA protocol.

A lot has been said about the blocking nature of XA (Open Group 1992) and two-phase commit in classic ACID transaction technology – we will not repeat that here. Suffice it to say: any successful transaction technology for SOA should avoid the distributed locks associated with XA. The most natural way of doing this is with TCC (Pardon 2009). Instead of introducing long-running ACID transactions, this allows us to use multiple, short-lived ACID transactions for each of the resource state transitions triggered by the "try", "cancel" and "confirm" events (Fig. 23.2). In addition to avoiding lots of problems, service-specific confirm and cancel logic are

also natural with respect to the business model of the service provider. This in turn means that transaction models embracing these will be less invasive and therefore more likely to be used.

Protocols

We will now introduce a set of protocols that ensure transactional correctness in REST systems. Let's start by defining the transaction a bit more formally:

Definition 1. A REST-based transaction T (e.g., booking a composed flight) is a number of invocations R_i (e.g., booking individual flights) across RESTful services S_i (e.g., swiss.com and easyjet.com) that need to either confirm altogether or cancel altogether. In other words: either all R_i succeed via an explicit confirmation $R_{i,confirm}$ (e.g., by paying for the flight), or all R_i cancel but nothing in between.

The Happy Path

1. A transactional workflow T goes about interacting with multiple distinct RESTful service APIs S_i
2. Interactions R_i may lead to a state transition of the participating service S_i identified by some URI – this URI corresponds to $R_{i,confirm}$
3. Once the workflow T successfully completes, the set of confirmation URIs and any required application-level payload is passed to a transaction service (or coordinator)
4. The transaction service then calls all of the $R_{i,confirm}$ with an idempotent `PUT` request on the corresponding URIs with the associated payloads

The protocol (Fig. 23.3) guarantees atomicity because each participating service receives a consistent request to either cancel or confirm. All participating services terminate their business transaction in the same way.

Note that we assume that $R_{i,confirm}$ is idempotent. In REST, this is a natural assumption to make. In practice, this means that the confirmation URI is called with a `PUT` or `DELETE` method – the particular choice depending on the contract defined by S_i and known to the workflow application, Fig. 23.4 illustrates this in the context of the running example.

Recovery Protocol

The basic protocol is very simple so it is natural to ask how this can work even in the presence of failures and recovery. Recovery is outlined below. We assume that each party is able to restore its own durable state, so we focus on the recoverability of the atomicity property across all parties.

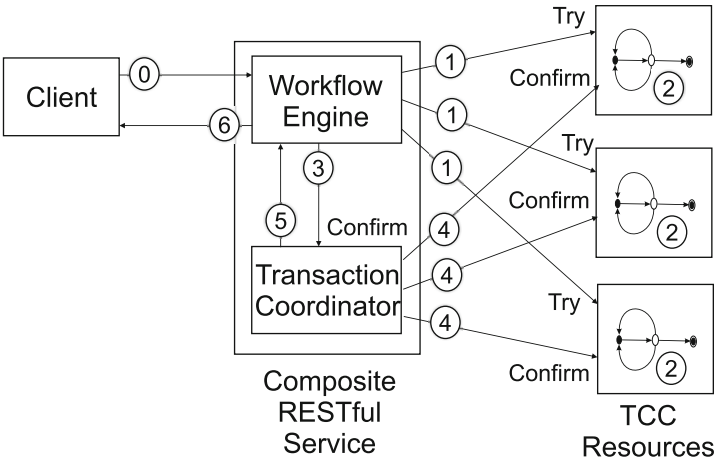

Fig. 23.3 Protocol architecture for the happy path

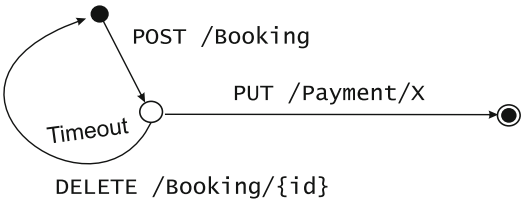

Fig. 23.4 Example of a flight reservation resource complying with the TCC pattern

Defining Recovery

For practical purposes, we define recovery as follows:

1. Checking the state of a transaction after node failure followed by restart, or
2. Checking the state of a transaction triggered by timeout

Recovery is something that is performed by the coordinator service as well as the participant services. For the coordinator this is expected since it intends to recover the transaction T that it knows about. For a participant, recovery also happens naturally: although a participant is not aware about T (following the loose coupling requirement), a participant service will want to release its reserved resources at the earliest possible time (as required by the business-level service contract).

Participant Service Recovery

Each participating service S_i does the following:

1. For recovery before step 2, do nothing.
2. For recovery after step 4: do nothing.
3. For recovery in between steps 2 and 4: execute $R_{i,cancel}$ autonomously (This can be triggered by a timeout).

Coordinator Recovery

Like the participant service, we assume that the coordinator service is capable of restoring its durable state. Consequently, we focus on the recoverability of the overall atomicity. The coordinator has a slightly more complex job than the participants, because it has to make sure that all the participants will eventually arrive at the same end state for the transaction T. In particular, step 4 actually involves multiple participants so a failure during step 4 could be problematic[1]. We propose a naive protocol here, and leave optimisations to future work.

1. For recovery before step 2, do nothing.
2. For recovery between steps 2 and 4: do nothing.
3. For recovery after step 4: do nothing.
4. For recovery during step 4: retry $R_{i,confirm}$ with each participating service S_i. Since $R_{i,confirm}$ are performed using idempotent methods, they may be retried as many times as necessary. Note: this requires the coordinator to durably log all participant information before starting step 4.

Discussion

This section presents reflections on the proposed protocols. In particular, we show that they can guarantee atomicity even in the event of failures and outline the known limitations of the approach.

Atomicity Guaranteed even with Failures

Even if there are arbitrary failures, we still preserve atomicity – eventually. In other words: given enough time, the global transaction T will be confirmed everywhere, or cancelled everywhere, or nothing will have happend in the first place. More precisely: either all R_i are confirmed, or all are cancelled. In order to prove this, we take a closer look at the protocol steps from the point of view of the coordinator. Here is our proof:

1. If there are no failures, then steps 1–4 run through and each R_i will have been confirmed.
2. For any failures before step 2, no R_i exists, meaning that nothing has happened.
3. For any failures during or after step 2 but before step 4: all R_i will eventually be cancelled autonomously by each S_i (since nothing has been confirmed yet).

[1] Especially because the participants are not aware that they are part of a transaction

4. For any failures during step 4: the coordinator will retry each $R_{i,confirm}$ until it succeeds. Because confirmation is idempotent, this will eventually succeed (note: there is one caveat here – discussed next).
5. For any failures after step 4: all $R_{i,confirm}$ have been done, so we already have atomicity and no action is required.

The Exception that Confirms the Rule: Heuristics

There is one weak spot in our proof of atomicity: during step 4, some service S_i may time out and cancel on its own, while the coordinator is performing confirmation. In the worst case, this means that some participants confirm whereas others cancel on their own – effectively breaking atomicity. We call this a *heuristic exception* for reasons outlined in the following.

Perfection does not Exist

There has been a lot of interesting work related to atomicity, and the more general problem of distributed agreement, and the most important result is that a perfect solution is not possible (Fischer 1985). In practice, this means that there is always the possibility that at least one participating service/node is unaware of the outcome of the "global" distributed transaction - be it with our TCC protocol for RESTful Web services or with classic, ACID, XA-style transactions.

The practical consequence is that one or more nodes can remain "in-doubt" about the global result of one or more business transactions that they are participating in. For instance, flight reservations may never complete because payment never arrives (either due network failures, node failures or both).

This is not specific to REST or WS-*, it exists in any networked environment: there is no perfect protocol for distributed agreement. This is a limitation one has to live with (and one of the drivers behind the CAP theorem Brewer 2000).

Enter Heuristics

The bottom line is that perfect atomicity may not be possible sometimes, and we need a practical way of dealing with such scenarios (just like workflow-based solutions do). We propose a simple way based on the "heuristic exceptions" known from the industry's two-phase commit protocol families (such as OTS Ram et al. 1999).

In practice, most industrial distributed two-phase commit (2PC) technologies recognize that similar anomalies may happen. In order to avoid that a participant

remains in-doubt about the outcome, these protocols allow the participants to timeout and unilaterally terminate their part of the a global transaction with a so-called "heurisitic decision" (e.g., heuristic rollback).

Our Protocol Compared to Two-Phase Commit

Once a participant completes R_i, it can be considered in-doubt: all its durable state changes are on disk, and the only remaining thing is the pending confirmation $R_{i,confirm}$ on behalf of T. If the participant decides to time-out then this is similar to what classical two-phase commit calls a heuristic rollback. The default way of handling this is very similar: we make sure that the coordinator logs this fact on behalf of T and assume that this will be reported in some implementation-specific way to allow for out of band manual resolution of the inconsistency by a human operator.

Advantages of our Protocol Compared to Classical Two-Phase Commit

One big advantage our protocol offers (compared to classical heuristic cases) is the fact that it offers higher-level semantics and does not hold low-level database locks. In-doubt participants do not block any other work other than the one affected by the business resources they reserve on behalf of T. When a heuristic cancel is done by S_i, the consequences are well-defined and known to the business: it corresponds to a unilateral breach (by S_i) of the contract entered into with the execution of R_i. Both the coordinator of T and the site administrators at S_i can use the high-level information to manually resolve the inconsistency. Contrast this to classical transactions, where heuristic exceptions are very low-level error conditions with vague impact and little context information. In this way, our protocol embraces the fact that distributed agreement between businesses is challenging due to the inherent limitations of distributed agreement and the CAP theorem.

Optimisations and Future Work

We have presented a simple protocol that ensures atomicity in at least as many cases as ACID transactions do, without the restrictions. However, there is a lot of room for optimisation. We can see at least the following things to refine:

1. Optimising the basic protocol with coordinator-driven cancellation in addition to confirmation. This allows the application/workflow to signal failures early, so that participating services do not have to time out. This in turn minimises resource contention.

2. Optimising timeout management by the coordinator in order to minimize the occurrence of heuristic exception cases. For instance, the coordinator could inquire (GET) with each participant to discover the remaining timeout before attempting to confirm. If the remaining timeout is below a threshold, then the coordinator might refuse to even start confirmation.
3. Optimising the handling of heuristic exceptions if they do happen. For instance, the coordinator could inquire at each participating service to find out more about what to do, or a management-by-exception type of workflow could be triggered that requires human intervention at the workflow end. This sounds all the more interesting because it is backed by the way that real businesses work today.
4. Our basic assumptions could be weakened. For instance, it might be that some service providers do not hold reservations. Likewise, it might be that some requests cannot fail under normal circumstances (like read-only `GET` requests). Further research along these lines, will help to widen the applicability of transactions over RESTful APIs which do not fully comply with the Try-Cancel/Confirm pattern.

Related Work

RESTful Service Composition

REST is widely perceived as an emerging lightweight technology for building Web services (Richardson and Ruby 2007). The properties of the REST architectural style are meant to enable *serendipitous reuse by means of composition* (Vinoski 2008).

The idea of RESTful service composition has also been explored in the Bite project (Rosenberg et al. 2008), or with the BPEL for REST extensions (Pautasso 2008). Also, Xu et al. (2008); Pautasso (2009) proposes to use workflow languages for composing RESTful services. All of these contributions to do not explicitly address the requirement for transactional composition of RESTful services.

RESTful Transaction Models

In addition to several threads on the *rest-discuss* mailing list, summarized by Little (2009), the problem of transactional interactions for RESTful services has started to attract some interest also in the research community. For example, an approach to RESTful transactions based on isolation theorems has been recently proposed in Razavi et al. (2009). The RETRO (Marinos et al. 2009) transaction model also complies with the REST architectural style.

REST-*

The recently appeared book "REST in Practice" (Webber et al. 2010) also has a dedicated chapter discussing transaction support for REST. The approach seems similar to what REST-* is trying to accomplish, with the same drawback of tight coupling due to, among other things, a transaction context going all around.

More in detail, the JBoss REST-* initiative aims at providing various QoS guarantees for RESTful Web services, in much the same way as WS-* has done for Web services by creating a "stack" of agreed upon best practices and standards for REST middleware. In its attempt at offering transactions, REST-* follows an approach that is reasonably close to TIP and WS-AT: a context is added to each invocation in order to make the invocation transactional. The receiving service has to understand that context in order to participate in the transaction outcome. This leads to tight coupling, something that we have tried to avoid.

ATOM Pub/Sub

Another common approach for reaching distributed agreement in REST uses a publish/subscribe mechanism based on feeds wherein the "transaction coordinator" publishes updates on the "outcome" of the transaction, and each participant then listens for any updates it might be interested in. This is certainly technically feasible, however it assumes that each participant knows the right feed that should be subscribed to, and understands the semantics of the updates being published by the coordinator. In our solution, the participants do not have to know anything besides their own business contract (API). Thus, we believe our approach introduces less coupling than this one.

Also, a publish/subscribe mechanism implies that the coordinator has no direct means of asking a participant service about its final outcome (taking into account any heuristic decisions it may have taken after a timeout). This seems a bit awkward to us.

Distributed Transaction Technologies

This section provides some relevant background information on related transaction technologies/standards for Internet-scale systems and/or service-oriented architectures.

TIP

The TIP (Transaction Internet Protocol) was one of the first initiatives to offer reliability on the wider scale of the Internet (Vogler et al. 1999), and across different

service providers. It is based on the notion of a transaction context that is passed along with each request. The notion of such a context is far from ideal because it introduces tight coupling and limits the interoperability of the participants.

CORBA OTS

Within the CORBA ecosystem (Henning 2006), the OTS (Object Transaction Service) is a distributed transaction framework that (at least in theory) provides interoperability of transactions across CORBA objects and even across ORBs (Ram et al. 1999). It is used primarily in financial and telecom industries and it allows for a certain heterogeneity. However, as every system based on binary ORB protocols and bindings, CORBA/OTS cannot be directly reused in the domain of RESTful Web services.

WS-*

The WS-* stack would not have earned its fame if it did not offer some form of transaction support. A number of competing standards have been proposed (Zimmermann et al. 2007), but all of them were designed by committee. This implies that they all tend to be somewhat over-engineered, and above all they are driven by technology vendors (Tai et al. 2004) rather than by practical needs or demands. Consequently, their practical relevance is rather limited.

The two most common approaches are the following: WS-AT and WS-BT. We will discuss them starting from the assumption that the main value proposition of the WS-* technology stack lies in its intrinsic interoperability between heterogeneous platforms.

Web Service – Atomic Transactions (WS–AT) is the WS-* counterpart of the classical ACID transaction technologies. It offers distributed XA transactions over web service protocols.

As far as we know, this is the only transaction standard that enjoys real cross-vendor support from the bigger players like IBM and Microsoft. Unfortunately, this complex specification leads to tight coupling between participating sites. Configuration is not easy, especially if security is involved. Interoperability among existing implementations has also been difficult to achieve.

Web Service – Business Activity (WS–BA) is a compensation-based protocol that arose out of the BPEL world as a way to make BPEL engines coordinate compensation scopes across vendors/engines. It offers the possibility to “compensate” for unrightfully executed work with application-level callbacks. However, there is no notion of a business-level “confirmation” phase, which may be needed to address our requirements.

We do not know of many vendors who support this standard. Microsoft, for instance, does not. This makes the usefulness for interoperability rather limited and hence the relevance of this technology may be questionable.

XA Technology

The XA (Open Group 1992) specification defines an open, vendor-independent way of supporting distributed ACID transactions across back-end systems. It is the classical way of doing distributed transactions a distributed system – but due to tight coupling limitations it is too restrictive for service-oriented architectures and REST.

Try Confirm/Cancel

Try Confirm/Cancel (TCC) is a business-level protocol for distributed atomic transactions offered by Pardon (2009). The main difference with the previously described approaches is that the transactional events corresponding to cancel ("rollback") and confirm ("commit") are not defined by the needs of the middleware/database but rather by the application/business services[2]. This makes TCC a highly practical and business-oriented protocol, which – as we have shown in this Chapter – fits very well within the constraints of the REST uniform interface.

Although the current implementations by Atomikos are based on protocols such as RMI/IIOP and WS-*, the underlying ideas lend themselves very well to RESTful Web services, without the need to introduce coupling. In fact, applying TCC to REST allows to offer distributed transactions with services that are unaware of being part of such atomic transaction.

Conclusion

In this chapter, we propose a light-weight atomic transaction solution for REST based on applying the Try-Cancel/Confirm (TCC) pattern to the design of a RESTful Web service. The pattern fits with the business requirements of many service providers (e.g., e-Commerce sites) that need to participate within long running transactions and thus offer services allowing clients to issue requests which can later be canceled and have to be confirmed within a given timeout before they are carried out.

In addition to defining the business case for REST transactions, we have proposed a simple protocol to achieve atomicity among distributed resources that comply with the TCC pattern. We illustrated the protocol's behaviour with an example also showing that the resources involved in the transaction remain unaware of the transaction. Finally we have discussed how the protocol provides a loosely coupled solution to guarantee atomicity and consistency in the event of failures and outlined the known limitations (shared by all distributed agreement protocols) mainly due to heuristic timeouts.

[2] A similar idea (but lacking the "try" phase) was also proposed in the OASIS BTP proposal (Dalal et al. 2003), which was standardized but remains without any current implementations.

References

Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Eric A. Brewer. Towards robust distributed systems (abstract). In *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing*, page 7, Portland, Oregon, July 2000.

Sanjay Dalal, Sazi Temel, Mark Little, Mark Potts, and Jim Webber. Coordinating Business Transactions on the Web. *IEEE Internet Computing*, 7(1):30–39, January 2003.

Roy Fielding. *Architectural Styles and The Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

Yaron Y. Goland, E. James Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring — WebDAV. Internet RFC 2518, February 1999.

Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*, pages 132–141, Asilomar, CA, January 2007.

Michi Henning. The Rise and Fall of CORBA. *ACM Queue*, 4(5):28–34, June 2006.

Mark Little. *REST and transactions?*, 2009. http://www.infoq.com/news/2009/06/rest-ts.

Alexandros Marinos, Amir R. Razavi, Sotiris Moschoyiannis, and Paul J. Krause. RETRO: A Consistent and Recoverable RESTful Transaction Model. In *Proc. of the IEEE International Conference on Web Services (ICWS 2009)*, pages 181–188, Los Angeles, CA, USA, July 2009.

Open Group. Distributed TP: The XA Specification, February 1992.

Guy Pardon. *Try-Cancel/Confirm: Transactions for (Web) Services*, 2009. http://www.atomikos.com/Publications/TryCancelConfirm.

Guy Pardon and Gustavo Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 210–219, Cairo, Egypt, September 2000.

Cesare Pautasso. BPEL for REST. In *7th International Conference on Business Process Management (BPM08)*, Milan, Italy, September 2008.

Cesare Pautasso. Composing RESTful Services with JOpera. In *Proc. of the International Conference on Software Composition (SC09)*, pages 142–159, Zurich, Switzerland, July 2009.

Cesare Pautasso and Erik Wilde. Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design. In *Proc. of the 18th International World Wide Web Conference*, pages 911–920, Madrid, Spain, May 2009.

Prabhu Ram, Lyman Do, Pamela Drew, and Tong Zhou. Object Transaction Service: Experiences and Open Issues. In *International Symposium on Distributed Objects and Applications (DOA 1999)*, pages 296–304, Edinburgh, UK, September 1999.

Amir R. Razavi, Alexandros Marinos, Sotiris Moschoyiannis, and Paul J. Krause. RESTful Transactions Supported by the Isolation Theorems. In *ICWE'09*, pages 394–409, 2009.

Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, May 2007.

Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Kahalf. Composing RESTful Services and Collaborative Workflows. *IEEE Internet Computing*, 12(5):24–31, September-October 2008.

Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data Knowl. Eng.*, 51(1):59–79, 2004.

Steve Vinoski. Serendipitous Reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.

Hartmut Vogler, Marie-Luise Moschgath, Thomas Kunkelmann, and J. Grünewald. The Transaction Internet Protocol in Practice: Reliability for WWW Applications. IEEE Computer Society, Internet Workshop'99 (IWS'99), February 1999.

Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice*. O'Reilly, September 2010.

Xiwei Xu, Liming Zhu, Yan Liu, and Mark Staples. Resource-Oriented Architecture for Business Processes. In *Proc of the 15th Asia-Pacific Software Engineering Conference (APSEC2008)*, December 2008.

Olaf Zimmermann, Jonas Grundler, Stefan Tai, and Frank Leymann. Architectural Decisions and Patterns for Transactional Worlflows in SOA. In *Proc. of the 5th International Conference on Service-Oriented Computing*, Vienna, Austria, 2007.