Atomic Distributed Transactions: a RESTful Design

Guy Pardon
ATOMIKOS
guy@atomikos.com
http://www.atomikos.com

Cesare Pautasso
Faculty of Informatics, University of Lugano, Switzerland
c.pautasso@ieee.org
http://www.pautasso.info

ABSTRACT

The REST architectural style supports the reliable interaction of clients with a single server. However, no guarantees can be made for more complex interactions which require to atomically transfer state among resources distributed across multiple servers. In this paper we describe a lightweight design for transactional composition of RESTful services. The approach – based on the Try-Cancel/Conf rm (TCC) pattern – does not require any extension to the HTTP protocol. The design assumes that resources are designed to comply with the TCC pattern and ensures that the resources involved in the transaction are not aware of it. It delegates the responsability of achieving the atomicity of the transaction to a coordinator which exposes a RESTful API.

Categories and Subject Descriptors

K.4.4 [**Electronic Commerce**]: Distributed Commercial Transactions; H.2.4 [**Systems**]: Transaction processing

Keywords

RESTful Web services; REST; Web API Design; Atomicity; Atomic Distributed Transaction Protocol

1. INTRODUCTION

Reliability of single client-server interactions is considered as a primary concern by the REST architectural style [1]. This is achieved through the uniform interface semantics of idempotent methods (e.g., `GET`, `PUT`, `DELETE` in HTTP) so that any failure during these interactions can be addressed by simply retrying the request. However, no guarantees can be made for complex interactions which atomically transfer state among multiple resources distributed across multiple RESTful Web services [11]. For example, when a client interacts with more than one RESTful APIs for f ight reservations, we want to ensure that all requests are performed atomically to complete the reservation of all f ights as a single step.

The goal of this paper is to describe a simple solution which f ts the following design constraints: 1) Using a lightweight transaction model (e.g., ATOMIKOS TCC [8]) to minimize interoperability risks; 2) Avoiding extensions to the HTTP protocol to maximize adoption; 3) Deploying the transaction coordinator as a RESTful service (as motivated in the remainder of this paper); 4) Keeping the participants unaware that they are part of a transaction (shipping transaction contexts has shown to be major pain point of distributed transactions).

WWW'14 Companion,, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04. http://dx.doi.org/10.1145/2567948.2579221.

The problem about how to transparently deal against failure scenarios within REST compositions, where state needs to be atomically transferred between more than two services, is an important one. The solution makes it possible to group multiple RESTful interactions and treat them as a single logical step, as well as to ensure that it is possible to guarantee the consistency of a set of resources which belong to multiple RESTful Web services that are deployed on different Web servers. Whereas solutions have been proposed to batch interactions affecting multiple resources provided by a single server (e.g., WebDAV's explicit locking methods [3], or the transactions as a resource approach from [11, p.231]), these are not applicable to interact with multiple resources distributed across multiple services.

This paper's contribution focuses on addressing the atomicity property [4] of distributed transactions across RESTful Web services *in a simple way* [6]. This already addresses 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 where clients may need to atomically perform multiple purchases, or, more in general, atomically change the state of a set of distributed resources.

2. ASSUMPTIONS

The protocol we follow is based on the following assumptions - without which there is no practical need for the outlined solution.

Assumption 1 (A_1): A business transaction T is sending requests R_i to different RESTful services S_j. Each participant S_j is autonomous and loosely-coupled with T and the other S_j.

Example: T represents a f ight reservation across two airlines, consisting of a request R_1 to book a f rst f ight at airline S_1 and a connecting f ight R_2 at airline S_2.

A_1 is important since it distinguishes our solution from existing techniques like session-based optimistic locking (a standard industry practice that works across one site only [2, p.416]). Note that for simplicity, we assume no ordering among R_i (no explicit f rst or last request).

Assumption 2 (A_2): Any R_i of T can fail in a non-transient way.

Example: booking the connecting f ight R_2 may fail due to lack of available seats.

A_2 emphasizes the point that if nothing can ever fail then idempotence is all that is needed for atomicity when each request R_i is retried until all have succeeded. Idempotence can help resolve transient, technical failures but not fundamental failures such as lack of business resources to comply with the request.

Assumption 3 (A_3): T needs to be atomic, i.e., T needs to either happen entirely (all $R_i \in T$ succeed) or not at all. In the latter case, the end result needs to be as if none of the R_i executed in the f rst place.

Example: you want to book the entire flight (R_1 and R_2) or none at all. If only part of the flight would be booked, you would be charged for an incomplete trip that would lead you to the wrong destination.

A_3 states the important requirement that the outcome of the whole transaction is what matters.

Assumption 4 (A_4): S_i temporarily reserves resources on behalf of R_i in T.

Example: airline S_1 holds a seat reservation for the duration of T, but will not keep the seat reserved forever: either T succeeds (and S_1 bills the customer) or T fails (and S_1 releases the seat for another customer to book).

Corollary 1 (C_1): Each R_i taking part in T may need to be cancelled after it has been executed. Consequently, each R_i needs to have a cancellation event $R_{i,cancel}$ associated with it.

From $A_1 + A_2$ it follows that partial transactions can exist in case of failure(s)[1]. For instance, it is possible for R_2 to fail, leaving the effects of R_1. From A_3 it follows that R_1 then needs to be cancelled. S_1 is autonomous (by A_1) so this requires the existence of a cancel event $R_{1,cancel}$ to inform S_1 about this event. In this case, by invoking $R_{1,cancel}$, the system can ensure atomicity. The same holds for every R_i.

Corollary 2 (C_2): Each participant S_i will offer/await a confirmation $R_{i,confirm}$ for every R_i.

From A_4 we know that S_i reserves resources on behalf of T. From C_1 we know that R_i can be cancelled. S_i will thus await confirmation of R_i to signal that cancellation will no longer happen. Since only T can know when all its requests have successfully finished, it is the responsibility of T to trigger the confirmation.

Example: after flight reservation R_2 completes, also flight R_1 needs to be confirmed with S_1 (and by T) so that payment can be issued. Cancelling a confirmed flight may still be possible but could lead to cancellation fees.

3. PROTOCOL OUTLINE

The previous assumptions lead to this protocol:

1. A client goes about interacting with multiple RESTful service APIs following a given workflow.

2. Interactions may lead to a state transition of the service identified by some reservation URI. This URI can be later used to confirm or cancel it. If the service does not hear anything after some service-specific timeout, it will cancel autonomously.

3a. Once the workflow successfully completes, the set of reservation URIs is used to confirm the state transitions of the services with idempotent requests (e.g., `PUT`).

3b. If the workflow fails, the set of reservation URIs that have been collected until the failure occurs are used to signal each of the services with idempotent messages (e.g., `DELETE`) to cancel their state transition.

The protocol guarantees atomicity because if it stops before steps 3a/3b the result is a cancel performed independently by each participant after a timeout. Otherwise each participant receives an idempotent request for confirmation (step 3a)/cancellation (step 3b) sent during the final phase. A more detailed discussion covering most failure and recovery scenarios can be found in [9] and in the detailed design presented below.

As with every two-phase commit solution, heuristics are needed to deal with timeouts. To deal with them, we propose that in step 2, a timeout is specified after which the service will unilaterally cancel. If step 3 happens too late then this might result in 'heuristic' anomalies (i.e. the transaction atomicity was violated). In that case, human intervention is required to reconcile the state across all sites. In any case, the transaction service in step 3 is free to decide when it will attempt confirmation (i.e., it might conservatively abort the transaction if the participants are too close to expiring).

[1] As failures we abstract both business-level failures as well as technical failures, such as server crashes or network outages.

4. EXAMPLE

As we are going to show, we claim that the REST uniform interface is sufficient to comply with the assumptions required to implement the proposed protocol. Thus, it is possible to achieve distributed transactions over RESTful service APIs without any extension of the HTTP protocol, *if the services are designed to comply with the Try-Cancel/Confirm pattern.*

In the context of the running example, this pattern can be applied to the design of the RESTful flight reservation API as follows. Clients can inquire about the availability of flights at the URI: `/flight/{flight-no}/seat`. For example, the `GET /flight/LX101/seat` request will return a hyperlink to some of the available seats on the flight `LX101` or none if the flight is fully booked. The URI of the chosen seat can be forwarded with a `POST` request to the `/booking` URL, which will create a new booking resource by returning a hyperlink identifying it such as `/booking/{id}/`. The body of the request can contain payment information as well as a reference to the chosen flight and seat (i.e., `<seat href="/flight/LX101/seat/63F"/>`). Seats on a flight are only reserved for a limited amount of time, during which the client should confirm the reservation. The booking can be confirmed using a `PUT /booking/{id}` request or canceled with the corresponding `DELETE /booking/{id}` request.

The example illustrates that a RESTful service API compliant with the uniform interface constraints can make use of hypermedia design to support the interaction with clients following the Try-Cancel/Confirm pattern. In the examples, clients initialize the state of a reservation with a standard `POST` request, which returns a URI identifying the resource that has been initialized. Clients can use this URI to confirm the reservation (with `PUT`) or to cancel it (with `DELETE`). Additionally, the state of the newly created resource should be discarded if a confirmation is not received by the service within a given timeout (the duration of which can be discovered by the client with a `GET` request on the same hyperlink).

A possible successful run of the protocol guaranteeing the atomicity of multiple HTTP interactions could be summarized as follows:

```
1.  ⇒GET swiss.com/flight/LX101/seat
    ⇐200
1.  ⇒GET easyjet.com/flight/EZ999/seat
    ⇐200
    ⇒POST swiss.com/booking
2.  ⇐302
      Location:  /booking/A
    ⇒POST easyjet.com/booking
2.  ⇐302
      Location:  /booking/B
3a. ⇒PUT swiss.com/booking/A
    ⇐204
3a. ⇒PUT easyjet.com/booking/B
    ⇐204
```

The following shows a failed run of the composition, where the protocols will perform the cancellation of the successfully completed state transitions:

```
1.  ⇒GET swiss.com/flight/LX101/seat
    ⇐200
    ⇒POST swiss.com/booking
2.  ⇐302
      Location:  /booking/A
```

```
1.  ⇒GET easyjet.com/flight/EZ999/seat
    ⇐204 (No seat available)
3b. ⇒DELEte swiss.com/booking/A
    ⇐200
```

5. DETAILED PROTOCOL DESIGN

The section proposes an incremental presentation of our design decisions, motivations and trade-offs, based on a story-based approach. Wherever possible, we have kept the response body to a minimum (merely 204 status) in order to avoid the need for def ning ad-hoc representation media types that introduce more coupling than necessary.

5.1 The Basics: Cancel vs Conf rm

One of the properties of classical transactions is the guarantee that every change is temporary (subject to 'rollback') until the application explicitly indicates that everything is done and can be saved ('commit'). For REST, we think the same should be possible. However, there is no classical 'rollback' because we use service invocations rather than databases and their locking mechanism to achieve this. In TCC, the notion of 'rollback' is replaced by 'cancel'. Likewise, the notion of 'commit' is replaced by 'conf rm'.

5.1.1 Cancel

As an application developer, in case of failures, I want to revert changes across multiple, separate participants.

For simplicity (and just like in classical transaction systems), we have chosen the cancellation mechanism to be implicit and internal to each participant service: after some time-out, each participant will/should cancel (revert) it on its own. This way, without further notif cations, each participant service will eventually cancel and the global transaction will be cancelled by default. This greatly simplif es the failure semantics across multiple participant services.

Note that our notion of cancelling does not preclude any application-specif c recovery mechanisms. For instance, an e-commerce website probably allows reservations to be made with a `POST` request. If the reply gets lost, the user might still be able to verify if the reservation was done (e.g., via a `GET` to some shopping basket or equivalent resource representing the session state) and continue from there. We merely offer the extra option of cancelling as a last resort.

5.1.2 Conf rm

As an application developer, I want to conf rm as soon as I am done so that no participant will cancel afterwards

If by default everything will be cancelled, there needs to be a way to perform otherwise. In TCC, this is done via an explicit 'conf rm' request on the participant service(s) involved. In order to do this with REST, the minimal requirement is a URI addressing the resource to be conf rmed. Only the participant service can/should determine what that URI is - so it needs a way to communicate this URI towards the outside world. With this in mind, we designed the notion of the **participant reservation link**. The link URI (used to conf rm it) is associated with the expires attribute indicating when the participant itself will cancel autonomously. There is also some (f xed) meta-data about the protocol itself, useful to indicate the semantics of the link (the `tcc` link relation).

The participant reservation link could be embedded in a JSON payload as shown in the following example:

```
{ "participantLink": {
    "uri":"http://www.swiss.com/booking/A",
    "expires":"2014-01-11T10:15:54.261+01:00",
    "rel":"tcc",
  }
}
```

It is up to the participant to negotiate with the client an appropriate timeout duration. In the simplest case, reservations are guaranteed for a f xed amount of time. It is also possible to consider cases where the timeout depends on the client prof le, or an extended timeout may be granted for a fee. Since this feature affects the interface between the participant and the application, we do not explore it further in this paper.

The assumption of our design is that a participant reservation link is properly identif ed (hence the `tcc` link relation) so that conf rmation can be done with the following:

```
⇒PUT /booking/A HTTP/1.1
  Host: www.swiss.com
  Accept: application/tcc
⇐HTTP/1.1 204 No Content
```

Although this only shows how to conf rm one single participant, it does lay the foundation for our complete solution.

5.1.3 Timeout

As a participant, I want to keep the ability to time-out and cancel the reservation on my end. Clients should be informed that conf rmation is no longer possible.

Conf rmation requests might come too late, i.e. after the participant already timed out and cancelled on its own. This violates the intention of conf rmation and therefore should be communicated to the caller. The participant does this as follows:

```
⇒PUT /booking/A HTTP/1.1
  Host: www.swiss.com
  Accept: application/tcc
⇐HTTP/1.1 404 Not Found
```

5.2 Reusing The Hard Bits: Coordinator

With nothing more but the basics, distributed transactions are possible if they are managed by the application developer (much like the XA protocol enables ACID transactions). However, this is error-prone and diff cult to manage, because concerns like recovery and failure handling need to be taken into account. Also, it is important to avoid conf rmation attempts after one or more timeouts have happened, since this may lead to conf icting outcomes of the global transaction. All this is specialized logic that is hard to build on your own. Just like ACID transactions rely on a transaction manager to manage the XA intricacies, we introduce a similar component sharing the same responsibilities.

5.2.1 Transaction Coordinator

As an application developer, I want to reuse existing conf rmation logic so that I don't have to deal with failure recovery on my own

If the conf rmation logic is offered as a reusable component (the 'coordinator') then many concerns no longer need to be dealt with by the application developer. Also, the error scenarios that are most diff cult to test are abstracted away into a reusable and tested component that can be trusted. For these reasons, we developed a transaction coordinator component. Moreover, this component can also be delivered as a service - as explained next and shown in Figure 1.

5.2.2 Transactions as a Service

As an application developer, I want the coordinator to be a RESTful service so I can access it anywhere, anytime

What better way to make a component available to a REST application than by exposing it as a RESTful service itself? There are many advantages, some of them being:

- Integration into REST applications is, by definition, easy and natural since no other technical dependencies than REST itself are introduced.
- The service can be made available to any device, anywhere, anytime.
- The service can be deployed on a reliable environment, with good connectivity to the participants, while keeping the rest of the applications closer to their users (e.g., on mobile devices).
- Thanks to the simplicity of REST and since no HTTP extensions are required, the interoperability of the transaction coordinator with both applications and participant services is much easier to achieve.

5.2.3 Confirmation

The application developer now needs a way to invoke the coordinator service to perform the confirmation phase.

We chose the following very simple approach, where a set of reservation links is simply transferred to the coordinator service with an idempotent `PUT` request carrying a 'transaction' payload in the request body. The example shows the use of plain JSON, but any collection media type that can carry a set of links associated with a set of timestamps can do.

```
⇒PUT /coordinator/confirm HTTP/1.1
  Host: www.taas.com
  Content-Type: application/tcc+json
  Content-Length: 253

 { "transaction" : [ {
       "uri" : "http://www.swiss.com/booking/A",
   "expires" : "2014-01-11T10:15:54.261+01:00"
     }, {
       "uri" : "http://www.easyjet.com/booking/B",
   "expires" : "2014-01-11T10:15:54.261+01:00"
 } ] }
```

The coordinator will delegate the confirmation request to all participant(link)s included in the supplied transaction. If all goes fine then the following would be the typical response:

```
⇐HTTP/1.1 204 No Content
```

For simplicity, we did not attempt a transaction resource design: there is no separate resource that identifies the transaction. Should this be necessary and required then we can always add it later. For now, all that matters to us is validating the concept of transactions for REST with a working but minimal implementation.

5.2.4 Failed Confirmation

As an application developer, I want to know when the coordinator failed to confirm.

When the coordinator fails to enforce the confirmation, we need a way to communicate the problem back to the application. There are two classes of problems that are relevant:

1. Problems where the overall atomicity guarantees have been preserved: this happens if ALL participants have timed out or have been canceled by the time confirmation starts.
2. Everything else: this is where the overall transaction guarantees might not have been preserved. This happens if some participants timed out while others confirmed, or if some participants have become unreachable.

The corresponding solutions are like this:

1. If every participant timed out and cancelled: this is indicated by a '`404 Not Found`' error on behalf of the coordinator. It signals that, although confirmation was desired, cancellation happened instead. While this may not be desirable, it does still adhere to the atomic transactional semantics of all-or-nothing.

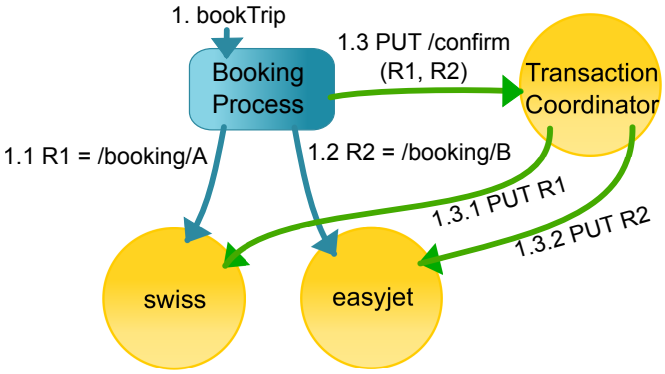

Figure 1: Transaction Coordinator delivered as a service.

2. Everything else: to signal conditions like these, the coordinator uses a '`409 Conflict`' status code and can return a detailed log, showing which of the given reservation links could be confirmed and which could not be confirmed. Note that it is the responsibility of the coordinator to minimize the number of failures in this class.

5.3 Recovery

5.3.1 Idempotent Confirmation (at the Participant)

As a coordinator, I want confirmation of the participant to be idempotent so I can retry confirmation after a failure or crash.

This is one of the main reasons why we chose to use `PUT` for confirmation. With the given design of the participant so far, we need no extra changes to support this.

All the coordinator needs to do is log the participant links of ongoing transactions in the confirmation phase. Recovery is needed in two typical cases:

1. The coordinator itself crashes: once it comes back up, it retries the remaining participant links for which it was confirming the transaction.
2. Any participant crashes, or (the equivalent) becomes unreachable due to network errors: the coordinator simply retries confirm requests.

5.3.2 Idempotent Confirmation (at the Coordinator)

As an application developer, I want confirmation by the coordinator to be idempotent so I can retry confirmation after a failure or crash.

Imagine that the application succeeds at doing all the work, at all participant service providers involved. At that time it would request the coordinator to confirm. If there are crashes or network failures then the response of the confirm request might get lost; this would leave the application in doubt about the outcome of the transaction.

According to the REST statelessness constraint, once the coordinator completes the confirm request, it should forget about it, so the application should hold its own state and should still remember the set of participants involved. Consequently, it can (and should) retry confirm requests when needed. This is why we chose to use `PUT` also for the coordinator's confirm requests. The coordinator will return the same response to subsequent confirmation requests involving the same participants.

5.4 Optimizations

The basic protocol can be optimized a bit for better resource usage. Indeed, if there are any application-level errors then it seems inefficient to simply let participants hold on to the required business

resources until they time out by themselves. So here, we present some optimizations.

5.4.1 Participant Cancellation

As a participant provider, I would like to be notif ed as early as possible when there is a need to cancel

The participant service is likely to reserve valuable business resources for the duration of the transaction. Should there be a need to cancel then it is very likely that the participant service wants to know about this well before it times out. Again, since we are talking about REST, the minimum requirement is a URI to follow for notifying the participant. For simplicity, we did not want to introduce an additional URI. Rather, we assume that the same URI representing the reservation resource that is used for conf rmation can optionally also be used for cancellation as follows:

```
⇒DELETE /booking/A HTTP/1.1
  Host: www.swiss.com
  Accept: application/tcc
⇐HTTP/1.1 204 No Content
```

Note that the actual response does not really matter since the cancellation request is merely a courtesy call on behalf of the application (developer). In its absence, the participant would cancel autonomously anyway.

The capability to cancel a participant explicitly is really optional in our design: any participant can choose to ignore it. If a participant provider does not support cancellation by the application then any DELETE request would simply produce:

```
⇐HTTP/1.1 405 Method Not Allowed
```

This does not affect overall correctness of the state, since the participant will time out and cancel autonomously anyway. Thus, the consistency of the distributed transaction is eventually preserved.

5.4.2 Coordinator Cancellation

As an application developer, I want to delegate the cancellation logic to the coordinator so I don't have to cancel those participant providers myself.

The following example shows how the application can cancel all the participants involved:

```
⇒PUT /coordinator/cancel HTTP/1.1
  Host: www.taas.com
  Content-Type: application/tcc+json
  Content-Length: 253

 { "transaction" : [ {
       "uri" : "http://www.swiss.com/booking/A",
   "expires" : "2014-01-11T10:15:54.261+01:00"
     }, {
       "uri" : "http://www.easyjet.com/booking/B",
   "expires" : "2014-01-11T10:15:54.261+01:00"
 } ] }
```

5.4.3 Failed Participant Cancellation

As a coordinator, I don't care if cancellation fails on the participant

The coordinator notif es the participant of cancellation, but ignores the result. This makes perfect sense, because cancellation is driven by the participant provider's needs to release reserved resource as early as possible. In effect, cancellation is merely a notif cation out of courtesy. There are a number of different reasons that support this decision:

1. Since explicit cancellation of a participant is really an optional operation, some participants may return 405 if they do not support this operation.

2. Since the participant may have timed out before the coordinator requests an explicit cancellation on it, it may return 404.

3. The participant URI does not really exist for some reason.

In all these cases, the overall transaction is cancelled everywhere (since no participant is ever conf rmed). Hence, all these errors can be safely ignored by the coordinator - making the protocol more comfortable to use because application developers need to worry less about error handling.

5.4.4 Failed Coordinator Cancellation

As an application developer, I don't care if cancellation fails on the coordinator

For the same reasons, the coordinator does not return any errors upon cancellation; instead, it always return 204 (including cases where some participant URI does not exist).

Corollary: cancellation by the coordinator is idempotent

This follows from the previous discussion: failed cancellations at both participants and the coordinator can be ignored. Hence we decided to use the `PUT` method for the coordinator's cancellation interface: it is idempotent and allows request body content (unlike `DELETE`).

5.5 MIME Types

5.5.1 Participant: `application/tcc`

The participant interactions require no request payload, nor do they return any response payload. So we chose this MIME type purely for indicating the semantics of conf rm/cancel expected by the client. We deliberately omitted any payload from the participant interactions, so implementations can be as simple as possible with minimal interoperability risks. There is no need for the participant to support anything like XML or JSON for that matter.

5.5.2 Coordinator: `application/tcc+json`

JSON seemed the best option to make the coordinator accessible to the largest range of applications. As we imply custom semantics with some of the attributes, this is ref ected in the MIME type.

6. DISCUSSION

We have presented a minimalistic protocol that offers the basics of atomicity guarantees for transactions spanning across multiple RESTful Web services. This section provides an overview of design issues that are still open to further ref nement and discussion.

6.1 Application-Level Errors: Cancel after Conf rm

For simplicity, the coordinator does not check nor prohibit the case where the application f rst conf rms a transaction and then later cancels the same transaction - for whatever reason. We consider this to be bad behavior on the account of the application, but checking for it would mean introducing new error codes on both the participant side and the coordinator side. We've tried that, and as a result we could no longer tolerate the cancellation of unknown participants, nor could we tolerate other types of participant failures. The resulting added complexity seemed too high to justify the corresponding gains so we've chosen not to reject such sequence of requests. It is thus the responsability of the application developer to avoid that conf rmed transactions are then cancelled at a later point in time.

6.2 Security

We did not consider security because we thought it is an orthogonal matter typically dealt with by HTTPS. Nevertheless, there may

be arguments in favor of more non-trivial solutions such as URI signing, OAuth and the like. Likewise, we assume that the participant reservation URIs are public URIs that can be forwarded by the application to the coordinator. This is common behavior on the open Web, where links are meant to be shared. However, if the participant will only allow the original client application to follow the reservation link, then additional work is needed for the application to delegate trust to the coordinator so that also this other component is allowed to follow the link to the participant. We consider this issue to be part of future work, based on the feedback we get from this first implementation.

6.3 Transaction Resource Model

So far, a transaction only exists explicitly as the request body of a stateless confirm/cancel request towards the coordinator. There is no RESTful resource for it yet. For now, this minimalistic design should be enough to get us the necessary feedback concerning the feasibility and desirability of our approach. Applications should be able to build a resource-ful model on top of this, and later versions of our API should be able to incorporate such additions.

6.4 Discovery of Coordinator API

Clients of the coordinators do not need to include hard-coded references to the confirm and cancel URIs of the coordinator service. Instead, hypermedia can be used to let them discover the actual URIs with a `GET` request on the coordinator root URI. Hyperlinks will be returned referring to the confirmation URI (with a link relationship `rel="confirm"`) and to the cancellation URI (with a link relationship `rel="cancel"`). The standardization of these link relationships with IANA is currently pending.

7. RELATED WORK

In addition to several threads on the *rest-discuss* mailing list, summarized by [5], the problem of transactional interactions for RESTful services has started to attract some interest also in the research community. A recent survey of RESTful transaction models has been published here [7]. Our approach shares with 2PC the challenge of achieving a distributed agreement, however we build upon the notion of reservation which fits directly into the business model of the participant service provider and does not require participants to deal with low-level details of running 2PC protocol rounds.

An informal proof of the protocol upon which the design presented in this paper is based was originally published in [9]. This paper adds the concept of having the transaction coordinator delivered as a service and presents a detailed RESTful design for its interface and a systematic discussion of its main use cases, including recovery scenarios. A browser extension that can intercept participant reservation URIs as the user navigates between different sites has been presented in [10]. The browser extension makes use of an embedded implementation of the coordinator to atomically confirm a distributed transaction implicitly recorded by tracking the navigation activities of the browser. Using the API design presented in this paper, it becomes possible to off-load the confirmation to the coordinator delivered as a service.

8. CONCLUSIONS

In this paper we presented a RESTful design based on applying the Try-Cancel/Confirm pattern to the design of a RESTful service. The pattern fits with the business requirements of many service providers that need to participate within long running transactions that do not require isolation. Thus, they offer services allowing clients to issue requests triggering state transitions (or resource reservations) which can later be canceled and have to be confirmed within a given time window. These 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.

There are currently two known implementations of the design (one in Java by ATOMIKOS[2] and another in JavaScript by the University of Lugano). Since our solution does not require any HTTP protocol extension, but can be seen more like a pattern, or a design best practice, we do not think it can be standardized per se. However, the design presented in this paper could grow to become the standard interface of a RESTful transaction coordinator delivered as a service. To achieve this, we plan to submit the MIME types and Link relation to IANA. Likewise, it would be beneficial to provide scaffolding in several languages and frameworks to make it easier to support the TCC pattern when building well-behaved participants. As previously mentioned, future work also involves dealing with security issues and extending the coordinator API to support persisting transactions as resources in addition to the current stateless approach.

While there are already many examples of e-commerce Web sites that provide users with the ability to reserve items for a given time, we expect similar functionality to be pushed in the corresponding Web APIs, which will then require an agreed upon mechanism for advertising the presence of participant links within response payloads. This will be critical to achieve adoption of our approach, so that atomic compositions of RESTful services can work on the World Wide Web.

9. REFERENCES

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

[2] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, November 2002.

[3] Y. Y. Goland, E. J. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP extensions for distributed authoring — WebDAV. Internet RFC 2518, Feb. 1999.

[4] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Proc.of the Seventh International Conference on Very Large Data Bases*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[5] M. Little. *REST and transactions?*, 2009. `http://www.infoq.com/news/2009/06/rest-ts`.

[6] T. Margaria and M. Hinchey. Simplicity in IT: The power of less. *Computer*, 46(11):23–25, 2013.

[7] N. Mihindukulasooriya, M. E. Gutiérrez, and R. G. Castro. Seven challenges for RESTful transaction models. In *Proc. of Fifth International Workshop on RESTful Design (WS-REST 2014)*, 2014.

[8] G. Pardon. *Try-Cancel/Confirm: Transactions for (Web) Services*, 2009. `http://www.atomikos.com/Publications/TryCancelConfirm`.

[9] G. Pardon and C. Pautasso. Towards distributed atomic transactions over RESTful services. In *REST: From Research to Practice*, pages 507–524. Springer, 2011.

[10] C. Pautasso and M. Babazadeh. The atomic web browser. *Poster at the 22nd International World Wide Web Conference (WWW 2013)*, pages 217–218, May 2013.

[11] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[2] `http://www.atomikos.com/Documentation/TccForRest`