Tcc For Rest

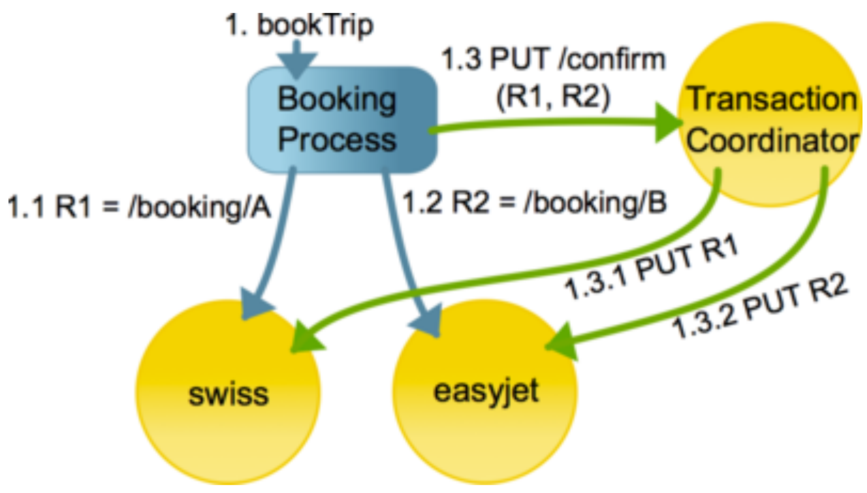
13 May 2016 - 06:25 | Version 20 | GuyPardon | REST API, TCC

The text below documents the Atomikos Composite Transactions for SOA REST API. It is the world's first transaction protocol that guarantees interoperability across vendors and platforms, thanks to its minimalistic approach and simplicity.

Table of Contents

Context Diagram

A sample TCC application is shown below: a transactional REST application (the booking process) goes about making related reservations across different, independent airlines as follows:



Example: booking connecting flights

1. (Step 1.1 in the picture) A flight reservation is made at "swiss". The provider ("swiss") will await confirmation for some timeout, and cancel automatically - and on its own - after that. Confirmation can happen only via the URI R1 that is returned to the booking process.
2. (Step 1.2 in the picture) A connecting flight is made at "easyjet", in the same way, and with confirmation URI R2.
3. (Step 1.3 in the picture) The booking process can now confirm both reservations together by invoking the coordinator service with both URIs. The coordinator service takes care of handling the confirmation and recovery logic on all services involved.

If there are any failures before 3, then everything will time out and cancel. If there are failures after 3, then everything is already confirmed. Any failures during 3 are handled by the coordinator service, including recovery after crashes and network failures. This yields transactional guarantees for REST services.

Roles

Our API consists of three complementary parts: the participant role, the coordinator role and the application.

- Participants are those third-party service providers that want their services to be TCC-aware (like the airline services in the example).
- The coordinator service is what we provide: a reusable service to manage the consistent confirmation (or cancellation) of a set of related participant service invocations (including recovery).
- Lastly, there is the application (the booking process). We impose no requirements on the application other than that it uses our coordinator in the ways documented here.

DISCLAIMER: Swiss and Easyjet are the trademarks of their respective owners.

For Service Providers: Participant API

Here we summarize the actual RESTful API from the participant perspective. Because participant instances are implemented by third-party providers, interoperability can only be achieved with a minimalistic, simple and clear design.

Participant Responsibilities

The participant manages the provider-specific state of a reservation of business resources. By default the reservation times out after a while, unless it is confirmed by the application (coordinator).

Required: Autonomous Timeout and Cancel

Every participant implementation MUST cancel autonomously after some internal timeout. Nothing is permanent until the participant receives confirmation.

Required: Participant Link

Every participant implementation MUST return participant link instances for an invocation that can be confirmed on its end. These links contain metadata such as the URI to invoke (for confirmation) and the expiration date/time when the participant will cancel on its own. Participant links are of the following form:

```
{"participantLink":
    {"uri":"http://www.example.com/part/123",
    "expires":"2014-01-11T10:15:54Z"
    }
}
```

The exchange of participant links is between the participant and the application, outside the context of the TCC protocol. Although our example suggests JSON, there is no real requirement on the data format of this exchange: this is entirely between the participant provider and the application developer to agree on. Other approaches, such as link headers can also be used.

Dates are in ISO8601 format.

Required: PUT to Confirm

The URI indicated in the participant link instances MUST support the PUT operation in order to confirm:

```
PUT /part/123 HTTP/1.1
Host: www.example.com
Accept: application/tcc
```

Note the MIME type of the request, indicating the expectations of the client about the semantics of the TCC protocol. It is typically the coordinator service that will call the participant to confirm.

Although the participant API has a specific MIME type, this type is only there to specify semantics. In particular, there is no request/response body content involved.

If all goes well, the participant response would be:

```
HTTP/1.1 204 No Content
```

If the confirmation request arrives after the participant has already timed out and cancelled on its own then the participant MUST return a 404 error:

```
HTTP/1.1 404 Not Found
```

Any other errors will trigger recovery logic in the coordinator service (typically in the form of retries until it gives up).

Optional: DELETE to Cancel

Each participant URI MAY optionally implement DELETE to receive explicit requests to cancel:

```
DELETE /part/123 HTTP/1.1
Host: www.example.com
Accept: application/tcc
```

The only expected return is:

```
HTTP/1.1 204 No Content
```

Any errors during cancel can be ignored and do not affect the overall transaction outcome (see below for some typical errors).

In case of an intermediate (internal) timeout/cancel by the participant itself, it is OK to return 404:

```
HTTP/1.1 404 Not Found
```

Since DELETE is really an optional operation, some participants may choose not to implement it. In that case:

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

This is perfectly fine in our overall design. Any others (such as, but not limited to, the MIME type not being understood) are also fine here.

Optional: GET for Failure Diagnostics

The participant service may implement GET to allow for failure diagnostics. In-line with our intent of being minimalistic, diagnostic features are (currently) outside the scope of our protocol itself and left to the application designers, so they can be tuned on a per-case basis.

For Application Developers: Coordinator API

The coordinator service is implemented by us and used by application developers. Therefore, we present the coordinator protocol from the point-of-view of a client of the RESTful interface as opposed to discuss the implementation internals of the coordinator.

Coordinator Responsibilities

The coordinator's core responsibilities are the following:

1. Confirm all participants when asked to do so.
2. Recover after failures of participant instances or the coordinator itself, in particular during the confirmation phase.

3. Intelligently use the supplied expiration date/time information to minimize the number of heuristic (i.e., problematic) transaction outcomes.
4. Determine the right error on problematic outcomes of confirmation.
5. Nice to have: easily allow cancellation of all participants when asked to do so.

PUT to Confirm

Use PUT to confirm a transaction with the coordinator service. A transaction is really only a collection of participant links:

```
PUT /coordinator/confirm HTTP/1.1
Host: www.taas.com
Content-Type: application/tcc+json
{    "participantLinks": [
        {
        "uri": "http://www.example.com/part1",
        "expires": "2014-01-11T10:15:54Z"
        },
        {
        "uri": "http://www.example.com/part2",
        "expires": "2014-01-11T10:15:54+01:00"
        }
    ]
}
```

Following this, the coordinator service will contact each participant and attempt confirmation. If all goes well then the result would be:

```
HTTP/1.1 204 No Content
```

If the request to confirm arrives too late - meaning all participants have timed out and cancelled already, then:

```
HTTP/1.1 404 Not Found
```

The worst that can happen is a mixed outcome where some participants confirmed, whereas others did not. This is indicated as follows:

```
HTTP/1.1 409 Conflict
```

Of course, the idea is to minimize the number of cases where this happens - which is one important part of the coordinator's responsibilities. If and when this happens, though, it is up to the application to inspect the affected participants - possibly via a GET request to each participant URI. Some sort of manual resolution would then be required, the exact details of which are intentionally left out of this specification to allow a best-fit approach per application.

PUT to Cancel

A cancellation request is similar to confirmation, except for the URI on which the coordinator is listening:

```
PUT /coordinator/cancel HTTP/1.1
Host: www.taas.com
Content-Type: application/tcc+json
{    "participantLinks": [
        {
        "uri": "http://www.example.com/part1",
        "expires": "2014-01-11T10:15:54Z"
        },
        {
        "uri": "http://www.example.com/part2",
        "expires": "2014-01-11T10:15:54Z"
        }
    ]
}
```

The only foreseen result is:

```
HTTP/1.1 204 No Content
```

Any other outcome can be safely ignored since by definition no participant has been confirmed, meaning eventually all work will be cancelled everywhere.

Additional Background Information

- Transactions for the REST of us
- Composite Transactions for SOA
- TCC as published at WS-REST 2014
- TCC slides from WS-REST 2014