REST-Atomic Transactions

2.0 draft 4

Version created 7 May 2010

Editors

Mark Little (mlittle@redhat.com)

Office 2004 Test Drive …, 24/12/09 13:25
Comment: Still to do:

Interposition.

Abstract

A common technique for fault-tolerance is through the use of atomic transactions, which have the well know ACID properties, operating on persistent (long-lived) objects. Transactions ensure that only consistent state changes take place despite concurrent access and failures. However, traditional transactions depend upon tightly coupled protocols, and thus are often not well suited to more loosely coupled Web based applications, although they are likely to be used in some of the constituent technologies. It is more likely that traditional transactions are used in the minority of cases in which the cooperating services can take advantage of them, while new mechanisms, such as compensation, replay, and persisting business process state, more suited to the Web are developed and used for the more typical case.

Table of contents

1 Note on terminology

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119 [1].

Namespace URIs of the general form http://example.org and http://example.com represents some application-dependent or context-dependent URI as defined in RFC 2396 [2].

2 REST-Atomic Transaction

Atomic transactions are a well-known technique for guaranteeing consistency in the presence of failures [3]. The ACID properties of atomic transactions (Atomicity, Consistency, Isolation, and Durability) ensure that even in complex business applications consistency of state is preserved, despite concurrent accesses and failures. This is an extremely useful fault-tolerance technique, especially when multiple, possibly remote, resources are involved.

Examples of coordinated outcomes include the classic two-phase commit protocol, a three phase commit protocol, open nested transaction protocol, asynchronous messaging protocol, or business process automation protocol. Coordinators can be participants of other coordinators. When a coordinator registers itself with another coordinator, it can represent a series of local activities and map a neutral transaction protocol onto a platform-specific transaction protocol.

2.1 Relationship to HTTP

This specification defines how to perform Atomic transactions using REST principles. However, in order to provide a concrete mapping to a specific implementation, HTTP has been chosen. Mappings to other protocols, such as JMS, is possible but outside the scope of this specification.

2.2 Header linking

Relationships between resources will be defined using the Link Header specification [4].

2.3 The protocol

The *REST-Atomic Transactions* model recognizes that HTTP is a good protocol for interoperability as much as for the Internet. As such, interoperability of existing transaction processing systems is an important consideration for this specification. Business-to-business activities will typically involve back-end transaction processing systems either directly or indirectly and being able to tie together these environments will be the key to the successful take-up of Web Services transactions.

Although traditional atomic transactions may not be suitable for all Web based applications, they are most definitely suitable for some, and particularly high-value interactions such as those involved in finance. As a result, the Atomic Transaction model has been designed with interoperability in mind. Within this model it is assumed that all services (and associated participants) provide ACID semantics and that any use of atomic transactions occurs in environments and situations where this is appropriate: in a trusted domain, over short durations.

Note, this specification only defines how to accomplish atomic outcomes between participations within the scope of the same transaction. It is assumed that if all ACID properties are required then C, I and D are provided in some way outside this scope of this specification. This means that some applications MAY use the REST-Atomic Transaction purely to achieve atomicity.

2.3.1 Two-phase commit

The ACID transaction model uses a traditional two-phase commit protocol [3] with the following optimizations:

- *Presumed rollback*: the transaction coordinator need not record information about the participants in stable storage until it decides to commit, i.e., until after the prepare phase has completed successfully. A definitive answer that a transaction does not exist can be used to infer that it rolled back.

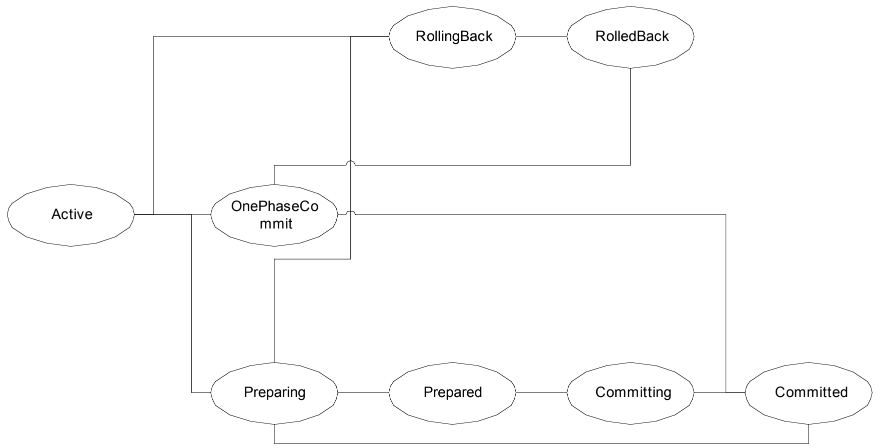
- *One-phase*: if the coordinator discovers that only a single participant is registered then it SHOULD omit the prepare phase.
- *Read-only*: a participant that is responsible for a service that did not modify any transactional data during the course of the transaction can indicate to the coordinator during prepare that it is a *read-only participant* and the coordinator SHOULD omit it from the second phase of the commit protocol.

Participants that have successfully passed the *prepare* phase are allowed to make autonomous decisions as to whether they commit or rollback. A participant that makes such an autonomous choice *must* record its decision in case it is eventually contacted to complete the original transaction. If the coordinator eventually informs the participant of the fate of the transaction and it is the same as the autonomous choice the participant made, then there is obviously no problem: the participant simply got there before the coordinator did. However, if the decision is contrary, then a non-atomic outcome has happened: a *heuristic outcome*, with a corresponding *heuristic decision*.

The possible heuristic outcomes are:
- *Heuristic rollback*: the commit operation failed because some or all of the participants unilaterally rolled back the transaction.
- *Heuristic commit*: an attempted rollback operation failed because all of the participants unilaterally committed. This may happen if, for example, the coordinator was able to successfully prepare the transaction but then decided to roll it back (e.g., it could not update its log) but in the meanwhile the participants decided to commit.
- *Heuristic mixed*: some updates were committed while others were rolled back.
- *Heuristic hazard*: the disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

2.3.2 State transitions

A transaction (coordinator and two-phase participant) goes through the state transitions shown:



There is a new media type to represent the status of a coordinator and its participants: application/txstatus, which supports a return type based on the scheme maintained at www.rest-star.org/… For example:

```
tx-status=TransactionActive
```

2.3.3 Client and transaction interactions

The transaction coordinator is represented by a URI. In the rest of this specification we shall assume it is http://www.fabrikam.com/transaciton-manager, but it could be any URI and its role need not be explicitly apparent within the structure of the URI.

2.3.3.1 Creating a transaction

Performing a POST on /transaction-manager with content as shown below will start a new transaction with a default timeout. A successful invocation will return 201 and the Location header MUST contain the URI of the newly created transaction resource, which we will refer to as transaction-coordinator in the rest of this specification. Two related URLs MUST also be returned, one for the terminator of the transaction to use (typically referred to as the *client*) and one used for registering durable participation in the transaction (typically referred to as the *server*). Although uniform URL structures are used in the examples, these linked URLs can be of arbitrary format.

```
POST /transaction-manager HTTP/1.1
From: foo@bar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 32
```

The corresponding response would be:

```
HTTP 1.1 201 Created
Location: /transaction-coordinator/1234
Link: /transaction-coordinator/1234/terminator;
rel=”terminator”
Link: /transaction-coordinator/1234/participant;
```

```
rel="durable participant"
```

Performing a HEAD on /transaction-coordinator/1234 MUST return the same link information.

```
HEAD /transaction-coordinator/1234 HTTP/1.1
From: foo@bar.com

HTTP/1.1 200 OK
Link: /transaction-coordinator/1234/terminator;
rel="terminator"
Link: /transaction-coordinator/1234/participant;
rel="durable participant"
```

Performing a POST on transaction-manager as shown below will start a new transaction with the specified timeout in milliseconds.

```
POST /transaction-manager HTTP/1.1
From: foo@bar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: --

timeout=1000
```

If the transaction is terminated because of a timeout, the resources representing the created transaction are deleted. All further invocations on the transaction-coordinator or any of its related URIs MAY return 410 if the implementation records information about transactions that have rolled back, (not necessary for presumed rollback semantics) but at a minimum MUST return 401. The invoker can assume this was a rollback.

Performing a GET on that /transaction-manager returns a list of all transaction coordinator URIs know to the coordinator (active and in recovery).

2.3.3.2 Obtaining the transaction status

Performing a GET on /transaction-coordinator/1234 returns the current status of the transaction, as described later.

```
GET /transaction-coordinator/1234 HTTP/1.1
Accept: application/txstatus+xml
```

With an example response:

```
HTTP/1.1 200 OK
Content-Length: --
Content-Type: application/txstatus

tx-status=TransactionActive
```

Performing a DELETE on any of the /transaction-coordinator URIs will return a 403.

2.3.3.3 Terminating a transaction

The client can PUT one of the following to /transaction-coordinator/1234/terminator in order to control the outcome of the transaction; anything else MUST return a 400. Performing a PUT as shown below will trigger the commit of the transaction. Upon termination, the resource and all associated resources are implicitly deleted. For any subsequent invocation then an implementation MAY return 410 if the implementation records information about transactions that have rolled back, (not necessary for presumed rollback semantics) but at a minimum MUST return 401. The invoker can assume this was a rollback. In order for an interested party to know for sure the outcome of a transaction then it MUST be registered as a participant with the transaction coordinator.

```
PUT /transaction-coordinator/1234/terminator HTTP/1.1
From: foo@bar.com
Content-Type: application/txstatus
Content-Length: --

tx-status=TransactionCommit
```

If the transaction no longer exists then an implementation MAY return 410 if the implementation records information about transactions that have rolled back, (not necessary for presumed rollback semantics) but at a minimum MUST return 401.

The state of the transaction MUST be Active for this operation to succeed. If the transaction is in an invalid state for the operation then the implementation MUST 403. Otherwise the implementation MAY return 200 or 202. In the latter case the Location header SHOULD contain a URI upon which a GET may be performed to obtain the transaction outcome. It is implementation dependent as to how long this URI will remain valid. Once removed by an implementation then 410 MUST be returned.

The transaction may be told to rollback with the following PUT request:

```
PUT /transaction-coordinator/1234/terminator HTTP/1.1
From: foo@bar.com
Content-Type: application/txstatus
Content-Length: --

tx-status=TransactionRollback
```

2.3.4 Transaction context propagation

When making an invocation on a resource that needs to participate in a transaction, the server URI (e.g., /transaction-coordinator/1234) needs to be transmitted to the resource. How this happens is outside the scope of this specification. It may occur as additional payload on the initial request, or it may be that the client sends the context out-of-band to the resource.

Note, a server SHOULD only use the transaction coordinator URIs it is given directly and not attempt to infer any others. For example, an implementation MAY decide to give the server access to only the root transaction coordinator URI and the participant URI, preventing it from terminating the transaction directly.

2.3.5 Coordinator and participant interactions

Once a resource has the transaction URI, it can register participation in the transaction. The participant is free to use whatever URI structure it desires for uniquely identifying itself; in the rest of this specification we shall assume it is /participant-resource.

2.3.5.1 Enlisting a two-phase aware participant

A participant is registered with /transaction-coordinator using POST on the participant Link URI obtained when the transaction was created originally:

```
POST /transaction-coordinator/1234/participant HTTP/1.1
From: foo@bar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: --

participant=/participant-resource/+
terminator=/participant-resource/terminator
```

Performing a HEAD on a registered participant URI MUST return the terminator reference, as shown below:

```
HEAD /participant-resource HTTP/1.1
From: foo@bar.com

HTTP/1.1 200 OK
Link: /participant-resource/terminator;
rel="terminator"
```

If the transaction is not Active then the implementation MUST return 403. If the implementation has seen this participant URI before then it MUST return 400. Otherwise the operation is considered a success and the implementation MUST return 201 and MAY use the Location header to give a participant specific URI that the participant MAY use later during prepare or for recovery purposes. The lifetime of this URI is the same as /transaction-coordinator. In the rest of this specification we shall refer to this URI as /participant-recovery (not to be confused with the /participant-resource URI) although the actual format is implementation dependant.

```
HTTP/1.1 201 Created
Location: /participant-recovery/1234
```

Note, in a subsequent draft we shall discuss how a participant can also register alternative terminator resources for the various operations used during the commit protocol. In this draft we assume that a uniform approach is used for all participants.

2.3.5.2 Enlisting a two-phase unaware participant

In order for a participant to be enlisted with a transaction it MUST be transaction aware in order that it can fulfill the requirements placed on it to ensure data consistency in the presence of failures or concurrent access. However, it is not necessary that a participant be modified such that it has a terminator resource as outlined previously: it simply needs a way to tell the coordinator which resource(s) with which to communicate when driving the two-phase protocol. This type of participant will be referred to as Two-Phase Unaware, though strictly speaking such a participant or service does need to understand the protocol as mentioned earlier.

During enlistment a service MUST provide URIs for prepare, commit, rollback and OPTIONAL commit-one-phase:

```
POST /transaction-coordinator/1234/participant HTTP/1.1
From: foo@bar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: --

participant=/participant-resource+
prepare=/participant-resource/prepare+
commit=/participant-resource/commit+
rollback=/participant-resource/rollback
```

Performing a HEAD on a registered participant URI MUST return these references, as shown below:

```
HEAD /participant-resource HTTP/1.1
From: foo@bar.com

HTTP/1.1 200 OK
Link: /participant-resource/prepare; rel=”prepare”
Link: /participant-resource/commit; rel=”commit”
Link: /participant-resource/rollback; rel=”rollback”
```

A service that registers a participant MUST therefore either define a *terminator* relationship for the participant or the relationships/resources needed for the two-phase commit protocol.

2.3.5.3 Obtaining the status of a participant

Performing a GET on the /participant-resource URL MUST return the current status of the participant in the same way as for the /transaction-coordinator URI discussed earlier. Determining the status of a participant whose URI has been removed is similar to that discussed for /transaction-coordinator.

2.3.5.4 Terminating a participant

The coordinator drives the participant through the two-phase commit protocol by sending a PUT request to the participant terminator URI provided during enlistment, with Prepare, Commit, Rollback or CommitOnePhase as the message content, i.e., requesting the state of the resource to be changed accordingly:

```
PUT /participant-resource HTTP/1.1
From: foo@bar.com
Content-Type: application/txstatus
Content-Length: --

tx-status=TransactionPrepare
```

If the operation is successful then the implementation MUST return 200. A subsequent GET on the URI will return the current status of the participant as described previously. It is not always necessary to enquire as to the status of the participant once the operation has been successful.

If the operation fails then the implementation MUST return 409. It is implementation dependant as to whether the /participant-resource or related URIs remain valid, i.e., an implementation MAY delete the resource as a result of a failure. Depending upon the point in the two-phase commit protocol where such a failure occurs the transaction MUST be rolled back. If the participant is not in the correct state for the requested operation, e.g., Prepare when it has been already been prepared, then the implementation MUST return 409.

If the transaction coordinator receives any response other than 200 for Prepare then the transaction MUST rollback.

Note, read-only MAY be modeled as a DELETE request from the participant to the coordinator using the URI returned during registration in the Location header, as mentioned previously, i.e., /participant-recovery. If GET is used to obtain the status of the participant after a 200 response is received to the original PUT for Prepare then the implementation MUST return 410 if the participant was read-only.

The usual rules of heuristic decisions apply here (i.e., the participant cannot forget the choice until it is told to by the coordinator).

Performing a PUT on /participant-resource/terminator with Forget will cause the participant to forget any heuristic decision it made on behalf of the transaction. If the operation succeeds then 200 MUST be returned and the implementation MAY delete the resource. Any other response means the coordinator MUST retry.

2.3.6 Recovery

In general it is assumed that failed actors in this protocol, i.e., coordinator or participants, will recover on the same URI as they had prior to the failure. If that is not possible them these endpoints SHOULD return a 301 status code or some other way of indicating that the participant has moved elsewhere.

However, sometimes it is possible that a participant may crash and recover on a different URI, e.g., the original machine is unavailable, or that for expediency it is necessary to move recovery to a different machine. In that case it may be that transaction coordinator is unable to complete the transaction, even during recovery. As a result this protocol defines a way for a recovering server to update the information maintained by the coordinator on behalf of these participants.

If the implementation uses the /participant-recovery URI described previously then a GET on /participant-recovery will return the original participant URI supplied when the participant was registered.

Performing a PUT on /participant-recovery will overwrite the old participant URI with the new one supplied. This will also trigger off a recovery attempt on the associated transaction using the new participant URI.

```
PUT /participant-recovery/1234 HTTP/1.1
From: foo@bar.com
Content-Type: application/x-www-form-urlencoded
Content-Length: --

new-address=URI
```

2.3.7 Pre- and post- two-phase commit processing

Most modern transaction processing systems allow the creation of participants that do not take part in the two-phase commit protocol, but are informed before it begins and after it has completed. They are called *Synchronizations*, and are typically employed to flush volatile (cached) state, which may be being used to improve performance of an application, to a recoverable object or database prior to the transaction committing.

This additional protocol is accomplished in this specification by supporting an additional two-phase commit protocol that enclosed the protocol we have already discussed. This will be termed the Volatile Two Phase Commit protocol, as the participants involved in it are not required to be durable for the purposes of data consistency, whereas the other protocol will be termed the Durable Two Phase Commit protocol. The coordinator MUST not record any durable information on behalf of Volatile participants.

In this case the Volatile prepare phase executes prior to the Durable prepare: only if this prepare succeeds will the Durable protocol be executed. If the Durable protocol completes then this MAY be communicated to the Volatile participants through the commit or rollback phases. However, because the coordinator does not maintain any information about these participants and the Durable protocol has completed, this SHOULD be a best-effort approach only, i.e., such participants SHOULD NOT assume they will be informed about the transaction outcome. If that is a necessity then they should register with the Durable protocol instead.

The Volatile protocol is identical to the Durable protocol described already. The only differences are as discussed below:

- It is an OPTIONAL protocol. An implementation that supports the protocol MUST show this when the transaction is created through a Link relationship: it returns an additional Linked resource whose relationship is defined as “volatile participant”. Services MUST use this URI when registering volatile participants.
- There is no recovery associated with the Volatile protocol. Therefore the /participant-recovery URI SHOULD NOT be used by an implementation.
- There can be no heuristic outcomes associated with the Volatile protocol.
- An implementation MAY allow registration in the Volatile protocol after the transaction has been asked to terminate as long as the Durable protocol has not started.
- There is no one-phase commit optimization for the Volatile protocol.

2.3.8 Checked transactions

Checked transactions have a number of integrity constraints including:

- Ensuring that only the transaction originator can commit the transaction.
- Ensuring that a transaction will not commit until all transactional invocations involved in the transaction have completed.

Some implementations will enforce checked behavior for the transactions they support, to provide an extra level of transaction integrity. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. A checked Transaction Service guarantees that commit will not succeed unless all invocations involved in the transaction have completed. Rolling back the transaction does not require such as check, since all outstanding transactional activities will eventually rollback if they are not told to commit

There are many possible implementations of checked transactions. One provides equivalent function to that provided by the request/response inter-process communication models defined by X/Open. It describes the transaction integrity guarantees provided by many existing transaction systems. In X/Open, completion of the processing of a request means that the service has completed execution of its invocation and replied to the request. The level of transaction integrity provided by a Transaction Service implementing the X/Open model of checking provides equivalent function to that provided by the XATMI and TxRPC interfaces defined by X/Open for transactional applications.

2.3.9 Statuses

Participants SHOULD return the following statuses by performing a GET on the appropriate /transaction-coordinator or participant URI:

- TransactionRollbackOnly: the status of the endpoint is that it will roll back eventually.
- TransactionRollingBack: the endpoint is in the process of rolling back.
- TransactionRolledBack: the endpoint has rolled back.
- TransactionCommitting: the endpoint is in the process of committing. This does not mean that the final outcome will be Committed.
- TransactionCommitted: the endpoint has committed.
- TransactionHeuristicRollback: all of the participants rolled back when they were asked to commit.
- TransactionHeuristicCommit: all of the participants committed when they were asked to rollback.
- TransactionHeuristicHazard: some of the participants rolled back, some committed and the outcome of others is indeterminate.
- TransactionHeuristicMixed: some of the participants rolled back whereas the remainder committed.
- TransactionPreparing: the endpoint is preparing.
- TransactionPrepared: the endpoint has prepared.
- TransactionActive: the transaction is active, i.e., has not begun to terminate.

The following status values are sent by the endpoints such as the coordinator to participants in order to drive them through the two-phase commit state machine:

- TransactionPrepare: the participant should attempt to prepare on behalf of the transaction.
- TransactionCommit: the recipient should attempt to commit. If the recipient is a participant and there has been no prepare instruction then this is a one-phase commit.
- TransactionRollback: the recipient should attempt to rollback.

3 References

[1] "Key words for use in RFCs to Indicate Requirement Levels," RFC 2119, S. Bradner, Harvard University, March 1997.

[2] "Uniform Resource Identifiers (URI): Generic Syntax," RFC 2396, T. Berners-Lee, R. Fielding, L. Masinter, MIT/LCS, U.C. Irvine, Xerox Corporation, August 1998.

[3] J. N. Gray, “The transaction concept: virtues and limitations”, Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[4] M. Nottingham, “HTTP Header Linking”, http://www.mnot.net/drafts/draft-nottingham-http-link-header-07.txt, June 2006.