REST-Atomic Transactions

2.0 draft 8

Version created 29 July 2013

Editors

Mark Little (mlittle@redhat.com)

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].

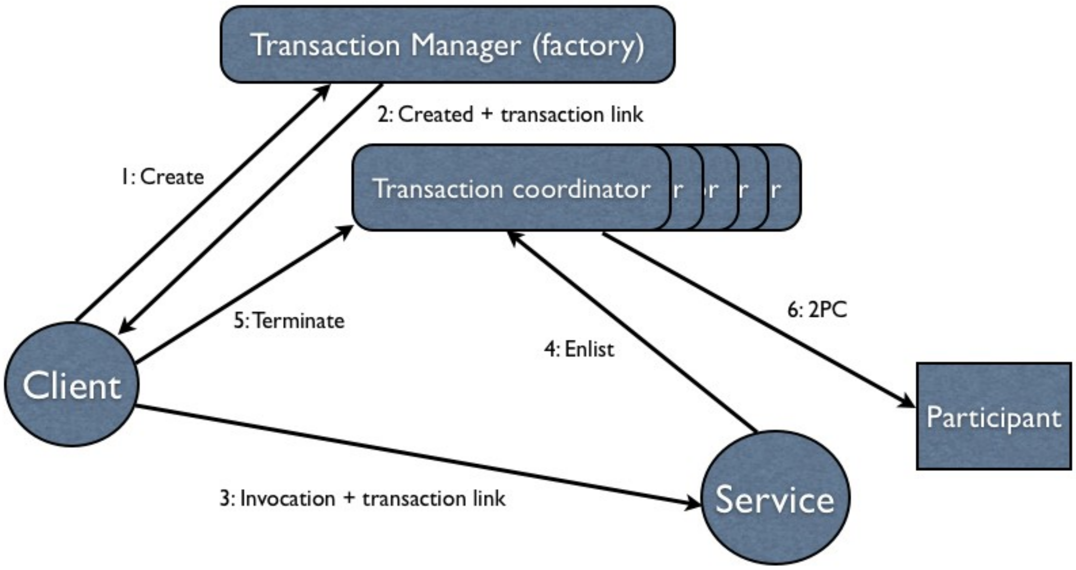
2.3 The protocol

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 suchh 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.

The following diagram illustrates the various components defined within this protocol. We shall discuss each of these in the remainder of this specification.



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.

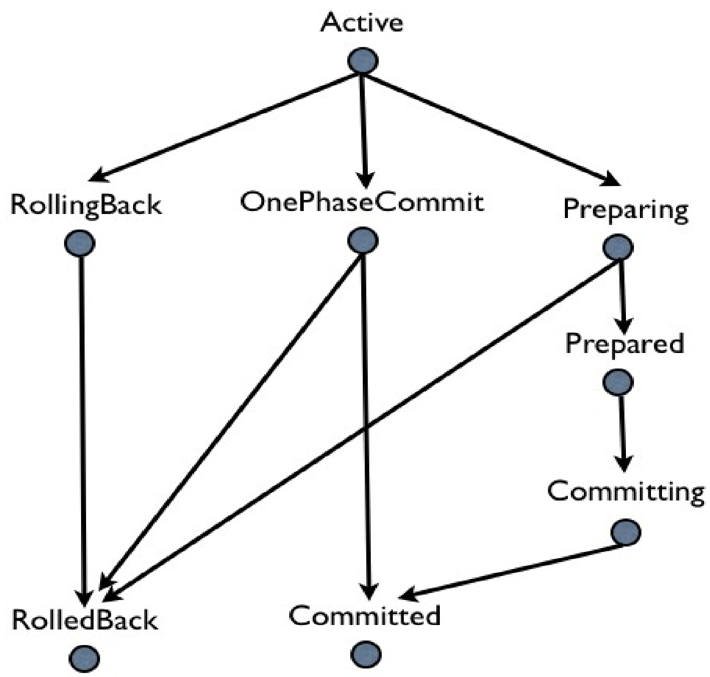
Participants that have successfully passed the *prepare* phase are allowed to make autonomous decisions as to whether they commit or rollback. A participant that makes such an autonomous choice *must* record its decision in case it is eventually contacted to complete the original transaction. If the 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 below. Note that non-atomic (heuristic) outcomes are not show on the diagram for simplicity, but are discussed in a later section:



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

~~tx-status~~txstatus=TransactionActive

The EBNF definition of this media type is:

<applicaton/txstatus> ::= "~~tx-status~~txstatus" "=" <tx-state>
<tx-state> ::=
"TransactionRollbackOnly" |
"TransactionRollingBack" |
"TransactionRolledBack" |
"TransactionCommitting" |
"TransactionCommitted" |
"TransactionCommittedOnePhase"
"TransactionHeuristicRollback" |
"TransactionHeuristicCommit" |
"TransactionHeuristicHazard" |
"TransactionHeuristicMixed" |
"TransactionPreparing" |
"TransactionPrepared" |
"TransactionActive" |
" TransactionStatusUnknown"

The text media type for a list of transactions (application/txlist) is simply a comma separated list of transaction URLs. In EBNF:

transaction url list ::= url { "," url}*
<url> ::= see RFC 1738

2.3.3 Client and transaction interactions

The transaction manager is represented by a URI (referred to as the transaction-manager URI). In the rest of this specification we shall assume it is http://www.fabrikam.com/transaction-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~~the transaction-manager URI with header 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. At least 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*). These are referred to as the transaction-terminator and transaction-enlistment URIs, respectively. Although uniform URL structures are used in the examples, these linked URLs can be of arbitrary format.

Note, an implementation MAY use the same URL for the terminator and participants.

```
POST /transaction-manager HTTP/1.1
From: foo@bar.com
```

The corresponding response would be:

```
HTTP 1.1 201 Created
Location: /transaction-coordinator/1234
Link:</transaction-coordinator/1234/terminator>;
rel="terminator",
</transaction-coordinator/1234/participant>;
rel="durable-participant",
</transaction-coordinator/1234/vparticipant>;
rel="volatile-participant"
```

An implementation MAY return a Link reference for volatile participants if it supports the OPTIONAL volatile two-phase commit protocol, which is described later in this specification.

Note, the coordinator does not have to be co-located with the transaction manager resource, nor does it need to have the same URL prefix.

Performing a HEAD on the transaction-coordinator URI 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”,
~~Link:~~</transaction-coordinator/1234/vparticipant>;
rel=”volatile~~–~~-participant”

Performing a POST on the transaction-manager URI 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: text/plain
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 404. The invoker can assume this was a rollback.

A failure during the POST request, such as a network partition, may mean that the initial response is not received. In this situation a client can retry the POST. Multiple transaction coordinators may be created as a result, but the client SHOULD only use one of them and the others will eventually timeout.

Performing a GET on the ~~/~~transaction-manager URI with media type application/txlist returns a list of all transaction-coordinator URIs known to the coordinator (active and in recovery). The returned response MAY include a link header with rel attribute "statistics" linking to a resource that contains statistical information such as the number of transactions that have committed and aborted. The link MAY contain a media type hint with value “application/txstatusext+xml”.

Performing a GET on the transaction-manager URI with media type application/txstatusext+xml returns extended information about the transaction-manager resource such as how long it has been up and all transaction-coordinator URIs.

2.3.3.2 Obtaining the transaction status

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

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

With an example response:

```
HTTP/1.1 200 OK
Content-Length: --
Content-Type: application/txstatus
Link:</transaction-coordinator/1234/terminator>; 
rel="terminator",
</transaction-coordinator/1234/participant>; 
rel="durable-participant",
</transaction-coordinator/1234/vparticipant>; 
rel="volatile-participant"

~~tx-status~~txstatus=TransactionActive
```

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

Additional information about the transaction, such as the number of participants and their individual URIs, MAY be returned if the client specifies the application/txstatusext+xml and the implementation supports that type, otherwise status 415 is returned (as per RFC 2616)..

2.3.3.3 Terminating a transaction

The client can PUT one of the following to the transaction-terminator URI ~~/transaction-coordinator/1234/terminator~~ in order to control the outcome of the transaction; anything else MUST return a 400 (unless the terminator and transaction URLs are the same in which case GET would return the transaction status as described previously). 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 PUT invocation, such as due to a timeout/retry, 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 404. 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~~txstatus=TransactionCommitted
```

The response body MAY contain the transaction outcome. 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 404.

The state of the transaction MUST be TransactionActive for this operation to succeed. If the transaction is in an invalid state for the operation then the implementation MUST return a 412 status code. Otherwise the implementation MAY return 200 or 202 codes. 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~~txstatus=TransactionRolledBack
```

2.3.4 Transaction context propagation

When making an invocation on a resource that needs to participate in a transaction, either the transaction-coordinator URI or the ~~enlisting~~transaction-enlistment URI (e.g., /transaction-coordinator/1234/participant) needs to be transmitted to the resource. This specification does not mandate a mechanism for propagation of this context information to the resource. However, the following OPTIONAL approaches are recommended.

- The URI is passed as a Link with the relevant service interaction.
- Services participating in the transaction return a Link to the client that can be used to register participation with the coordinator.

Note, a server SHOULD only use the URIs it is given directly and not attempt to infer any others.

2.3.5 Coordinator and participant interactions

Once a resource has the transaction or enlistment URI, it can register participation in the transaction. Each participant must be uniquely identified to the transaction in order that the protocol can guarantee consistency and atomicity in the event of failure and recovery. 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 and refer to it as the participant-resource URI.

2.3.5.1 Enlisting a two-phase aware participant

A participant is registered with the /transaction-coordinator using POST on the participant Link-enlistment URI obtained when the transaction was created originally. The request must include two link headers: one to uniquely identify the participant to the coordinator and one to provide a terminator resource (referred to as the participant-terminator URI) that the coordinator will use to terminate the participant. If the rel attributes of the link are not participant and terminator the implementation must return 400. Note, the following URIs are only examples, and an implementation is free to use whatever structure/format it likes:

```
POST /transaction-coordinator/1234/participant HTTP/1.1
From: foo@bar.com
Link:</participant-resource>; rel=”participant”,
</participant-resource/terminator>; rel=”terminator”
```

```
Content-Length: 0
```

Performing a HEAD on ~~a registered participant~~the participant-resource 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 TransactionActive when registration is attempted, then the implementation MUST return a 412 status code. 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 SHOULD 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 the transaction-coordinator URI ~~/transaction-coordinator~~. In the rest of this specification we shall refer to this URI as the participant-re~~v~~covery URI ~~/participant-recovery~~ (not to be confused with the ~~/~~participant-resource URI)~~ although the actual format is implementation dependent~~.

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

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 to 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) to communicate with 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.

Note, enlisting two-phase unaware participants is an OPTIONAL part of the specification. An implementation that does not support this MUST return 405.

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
Link:</participant-resource>; rel=”participant”,
</participant-resource/prepare>; rel=”prepare”,
</participant-resource/commit>; rel=”commit”,
</participant-resource/rollback>; rel=”rollback”,
</participant-resource/commit-one-phase>; rel=”commit-
one-phase”
```

Content-Length: 0

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”,
</participant-resource/commit-one-phase>; rel=”commit-one-phase”

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 URI~~L~~ 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 the ~~/~~transaction-coordinator URI.

GET /participant-resource~~/1234~~ HTTP/1.1
Accept: application/txstatus

With an example response:

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

~~tx-status~~txstatus=TransactionActive

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 the desired transaction outcome as the content (TransactionPrepared, TransactionCommitted, TransactionRolledBack or TransactionCommittiedOnePhase). For instance, here is how the prepare phase would be driven:

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

```
Content-Length: --

~~tx-status~~txstatus=TransactionPrepared
```

If PUT 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 PUT fails, e.g., the participant cannot be prepared, then the implementation MUST return 409. It is implementation dependentdependant 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, e.g., because we use presumed abort semantics, failures prior to the end of the prepare phase MUST result in a roll back. If the participant is not in the correct state for the requested operation, e.g., TransactionPrepared when it has been already been prepared, then the implementation MUST return 412.

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

After a request to change the resource state using TransactionRolledBack, TransactionCommitted or TransactionCommittedOnePhase, any subsequent PUT request MUST return a 409 or 410 code.

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., the /participant-recovery URI. 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 it made until it is told to by the coordinator).

Performing a DELETE on the /participant-resource URI 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; a subsequent PUT or GET request MUST return 410. 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. HTTP provides a number of options to support temporary or permanent changes of address, including 301 (Moved Permanently) and 307 (Temporary Redirect). If that is not possible then these endpoints SHOULD return a 301 status code or some other way of indicating that the participant has moved elsewhere. HTTP response codes such as 307 MAY also be used by the implementation if a temporary redirection is used.

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 recovering participant uses the /participant-recovery URI returned by the coordinator during enlistment then a GET on the /participant-recovery URI will return the participant resource and terminator as link headers ~~the original participant URI supplied when~~that the the participant was ~~registered~~used during the original registration.

Performing a PUT on the /participant-recovery URI will overwrite the old participant URI with the new one supplied. This operation is equivalent to re-enlisting the participant. This will also trigger off a recovery attempt on the associated transaction using the new participant URI. For example to update location URIs, a two phase aware participant would PUT the following document:

PUT /participant-recovery/1234 HTTP/1.1
From: **foo@bar.com**
Link:</new-participant-resource>; rel="participant",
</participant-resource/new-terminator>;
rel="terminator"
~~Content-Type: text/plain~~
Content-Length: ~~--~~0

~~new-address=URI~~

Similarly for a two phase unaware participant.

If, after performing the PUT request to the participant-recovery URI, the participant is not asked to complete (within an implementation dependent period) then it SHOULD reissue the PUT request.

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 where the transaction-coordinator sends a PUT request to the registered volatile-participant: only if this prepare succeeds will the Durable protocol be executed. The volatile-participant MUST indicate success by returning a 200 status code (any other code indicates failure). If the Durable protocol completes then this MAY be communicated to the Volatile participants through the commit or rollback phases. In this case the transaction-coordinator sends a PUT request to the registered volatile-participant with with the outcome in the request body (using content type application/txstatus). 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 Statuses

Resources MUST 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. If the recipient has already rolled back then it MUST return a 410 error code.
- 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. If the recipient has already committed then it MUST return a 410 error code.
- TransactionCommitted: the endpoint has committed.
- TransactionCommittedOnePhase: the recipient has committed the transaction without going through a prepare phase. If the recipient has previously been asked to prepare then it MUST return a 412 error code. If the recipient has already terminated, then it MUST return a 410 error code.
- 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.
- TransactionStatusUnknown: the status of the transaction is unknown

The statuses are also used to drive the two-phase commit protocol as discussed previously.

3 Security Model

The security model for atomic transactions builds on the standard HTTP security model. That is, services have policies specifying their requirements and requestors provide claims (either implicit or explicit) and the requisite proof of those claims. Coordination context creation establishes a base secret which can be delegated by the creator as appropriate.

Because atomic transactions represent a specific use case rather than the general nature of coordination contexts, additional aspects of the security model can be specified.

All access to atomic transaction protocol instances is on the basis of identity. The nature of transactions, specifically the uncertainty of systems means that the security context established to register for the protocol instance may not be available for the entire duration of the protocol. Consider for example the scenarios where a participant has committed its part of the transaction, but for some reason the coordinator never receives acknowledgement of the commit. The result is that when communication is re-established in the future, the coordinator will attempt to confirm the commit status of the participant, but the participant, having committed the transaction and forgotten all information associated with it, no longer has access to the special keys associated with the token.

There are, of course, techniques to mitigate this situation but such options will not always be successful. Consequently, when dealing with atomic transactions, it is critical that identity claims always be proven to ensure that coordinators maintain correct access control.

There is still value in coordination context-specific tokens because they offer a bootstrap mechanism so that all participants need not be pre-authorized. As well, it provides additional security because only those instances of an identity with access to the token will be able to securely interact with the coordinator (limiting privileges strategy).

The "list" of authorized participants ensures that application messages having a coordination context are properly authorized since altering the coordination context ID will not provide additional access unless (1) the bootstrap key is provided, or (2) the requestor is on the authorized participant "list" of identities.

4 Security Considerations

It is strongly RECOMMENDED that the communication between services be secured using HTTP security mechanisms. In order to properly secure messages, the body and all relevant headers need to be included in the signature. In the event that a participant communicates frequently with a coordinator, it is RECOMMENDED that a security context be established.

It is common for communication with coordinators to exchange multiple messages. As a result, the usage profile is such that it is susceptible to key attacks. For this reason it is strongly RECOMMENDED that the keys be changed frequently. This "re-keying" can be effected a number of ways. The following list outlines four common techniques:

- Attaching a nonce to each message and using it in a derived key function with the shared secret
- Using a derived key sequence and switch "generations"
- Closing and re-establishing a security context (not possible for delegated keys)
- Exchanging new secrets between the parties (not possible for delegated keys)

It should be noted that the mechanisms listed above are independent of the SCT and secret returned when the coordination context is created. That is, the keys used to secure the channel may be independent of the key used to prove the right to register with the activity.

Note, the content of Link header fields is not secure, private or integrity-guaranteed, and due caution should be exercised when using it. Use of Transport Layer Security (TLS) with HTTP [5] and [6]) is currently the only end-to-end way to provide such protection.

5 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.

[5] http://tools.ietf.org/html/rfc2818

[6] http://tools.ietf.org/html/rfc2817