RESTful Atomic Transactions

Mark Little
Red Hat
mark.little@jboss.com

Michael Musgrove
Red Hat
mmusgove@redhat.com

Bill Burke
Red Hat
bill.burke@jboss.com

ABSTRACT

Atomic transactions are a well-known technique for guaranteeing consistency in the presence of failures. The ACID properties of atomic transactions ensure that even in complex business applications consistency of state is preserved, despite concurrent accesses and failures. However, although this is an extremely useful fault-tolerance technique, it has yet to see widespread acceptance or adoption within the Web. Some believe that such capabilities are already provided within the Web whereas others think it is simply not possible to provide transactions with REST principles. In this paper we shall illustrate why transactions are needed and how they can be provided in a manner that fits within the Web's architectural principles. We shall also discuss how this protocol has been implemented using an open source project as well as with the JAX-RS standard, which is part of Java Enterprise Edition 6. Finally we will hint at an alternative approach to ACID transactions that we are working on currently.

Keywords
REST, transactions, atomicity, ACID, compensation transactions.

1. INTRODUCTION

Distributed systems pose reliability problems not frequently encountered in centralized systems. A distributed system consisting of a number of computers connected by a network can be subject to independent failure of any of its components, such as the computers themselves, network links, operating systems, or individual applications, and activities may take an indeterminate duration to execute. Decentralization allows parts of the system to fail while other parts remain functioning which leads to the possibility of abnormal behavior of executing applications.

The Web is the largest distributed system in history and suffers from failures that can affect both the performance and consistency of applications run over it. Fortunately for the majority of users these failures are a minor inconvenience and retrying a request later is a sufficient compensation approach. However, there are situations where retrying would not help, such as where coordination of an outcome across a number of endpoints (resources) has to be atomic, i.e., they either all do the work or none of them do the work.

Atomic transactions are a well-known technique for guaranteeing consistency in the presence of failures [1]. The ACID properties of atomic transactions (Atomicity, Consistency, Isolation, 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.

Consistency is especially important in a web application with dynamic servers. When users navigate a web application, they are viewing snapshots of the server state. If the snapshot is computed within a transaction, the state returned to the user is consistent. For many applications this is extremely important. Otherwise the inconsistent view of the data could be confusing to the user. Many developers have the incorrect perception that they do not need transactions if all they are doing is reading a database. However, if you are doing multiple reads and you want them to be consistent, then you need to do them within a transaction.

Furthermore, even in the simplest of system, a single user environment where all operations are idempotent, retrying requires the capability to remember the list of participating resources as well as the operation(s) that must be retransmitted, potentially many times. As we shall see, fortunately this is an inherent part of a transaction system, provided in a reliable manner such that it can tolerate its own failures as well as those that occur elsewhere within the environment.

In this paper we shall discuss a RESTful transaction protocol that we have developed as well as a corresponding implementation.

2. WHY REST?

REST has grown in popularity recently for a variety of reasons. Developers are attracted to the simplicity of the interfaces created. Since HTTP is such a ubiquitous protocol, developers get lightweight interoperability out of the box because most languages and platforms support both client and server interactions with their built-in HTTP support. REST also provides developers with a strong set of architectural guidelines and constraints. As developers explore these techniques, they are finding that their distributed interfaces become more decoupled, usable, and maintainable over time.

It is true that the Web and REST have progressed well without transactions. However, we believe that there are circumstances and particular applications where the use of transactions, or at least atomicity, would be beneficial. As we have evangelized REST to our customers and communities, we have found that a frequent question is: how can application developers leverage transactions?

This is often the result of having tried to do without transactions initially and found the resulting systems inadequate. Sometimes those users have come from backgrounds such as Java Enterprise Edition, where they expect such capabilities and have architected for them. Of course it could be that some of these applications were designed inappropriately and the apparent need for transactions would disappear through a careful redesign. However, this cannot account for all of these use cases. Furthermore, we believe from the input we have received from architects and users that a REST-based transaction protocol is an option that should be available for selection in certain situations.

To support this need, we decided to create a RESTful interface to our existing transaction manager. Beyond satisfying the requirements of our users and customers, we've found that a RESTful interface to transactions has a lot of benefits in the implementation of the protocol as a whole. All and all, it was a win-win scenario.

3. WHY NOT WS-TRANACTIONS?

There is a standard for transactions within the WS-* architecture [2]. WS-Transactions defines atomic and compensation based models and has demonstrated interoperability between all of the major transactions vendors. So the obvious question is why not simply use WS-Transactions? There are several reasons for this and we enumerate some of them below:

- The typical Web Services stack is often too large and complex for many users to want to invest time and effort in using. By leveraging HTTP as both a rich protocol and message format we can reduce the footprint at both the client and the server.
- The HTTP protocol already has a rich vocabulary that we can use to provide a more flexible protocol. For instance, we use Links to convey to clients different ways in which they can interact with the transaction manager.
- Out of the box the HTTP protocol conveys a set of guarantees that both the client and server must honour. For instance, if the coordinator PUTs a message to a participant and there is a network failure, it is possible to retransmit.
- The representational nature of REST allows us to support multiple transaction interaction formats simultaneously as well as let us evolve the protocol over time.

For these and other reasons we believe that an approach based on REST for integrating transaction systems and providing transaction capabilities to applications, is more suitable than WS-Transactions.

4. RESTFUL TRANSACTION PROTOCOLS

For over a decade some of us have been involved with the development of transaction protocols for the Web [3], but concentrating mainly on Web Services. However, in the past few years we have seen a shift in academia as well as industrial research and development from attempting to use protocols based around SOAP to those that more completely cooperate with the Web. Within JBoss we have seen an increased requirement from our users for transaction protocols that mirror the capabilities that are available in other enterprise middleware environments, but which are based on REST principles. Of course there are continuing arguments as to the benefit of incorporating transactions with REST [4], but from the perspective of our customers it seems meaningful and is certainly a requirement.

As such in the next sections we shall describe the atomic transaction protocol we have developed and implemented. Note that in order to provide a concrete mapping to a specific implementation, HTTP was chosen initially. Mappings to other protocols, such as JMS, are possible but have been left for future work.

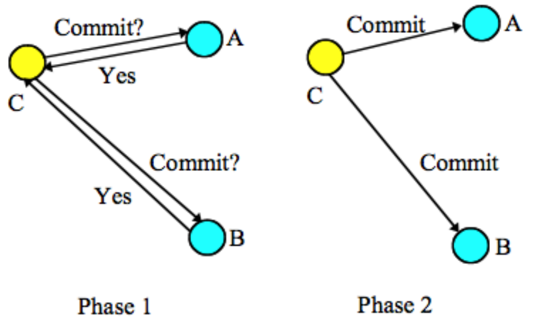
4.1 THE REST-ATOMIC TRANSACTIONS PROTOCOL

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 as it is for the users who have requested it. Business-to-business activities will typically involve back-end transaction processing systems either directly or indirectly and being able to tie together these environments is a common request in the enterprise middleware arena.

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 REST-Atomic Transaction model has been designed with interoperability in mind. However, this protocol 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 the protocol. This means that some applications may use the REST-Atomic Transaction purely to achieve atomicity. In fact this is consistent with many transaction protocols, such as the OTS [5] and WS-Atomic Transactions [6], which are also only concerned with implementing the consensus protocol necessary to achieve atomicity.

4.1.1 Reaching consensus

Traditional transaction systems use a two-phase protocol to achieve atomicity between participants, as illustrated below: during the first (preparation) phase, an individual participant must make durable any state changes that occurred during the scope of the transaction, such that these changes can either be rolled back or committed later once the transaction outcome has been determined. Assuming no failures occurred during the first phase, in the second (commitment) phase participants may "overwrite" the original state with the state made durable during the first phase.



The REST-Atomic Transaction (REST-AT) model uses a traditional two-phase commit protocol [7] 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 may 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 can omit it from the second phase of the commit protocol.

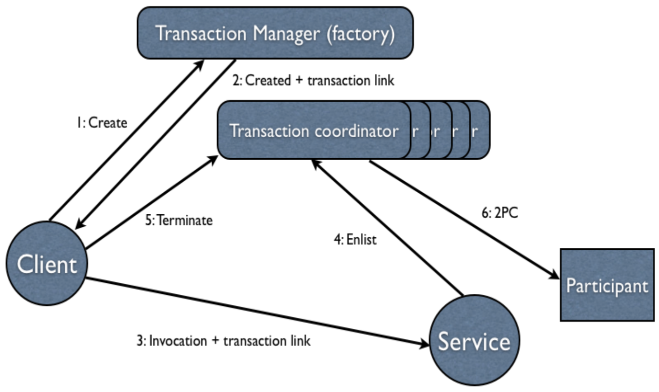
The fact that two-phase commit is a blocking protocol raises an important question: what happens if the coordinator fails? Normally participants that have passed the prepare state would remain blocked, potentially forever. Obviously that is not a suitable situation in practice. Most transaction protocols and implementations provide a way around this through *heuristic choices*.

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 [7]. Due to space limitations we shall not say anything more on heuristics except that the REST-AT protocol supports them.

Note, in what follows, relationships between resources are defined using the Link Header specification [8]. Furthermore, due to space considerations in the following discussion we omit some of the error handling aspects of the protocol.

4.2 ARCHITECTURE

The diagram below illustrates the various resources defined within the REST-AT protocol. We shall discuss each of these in the following sections.

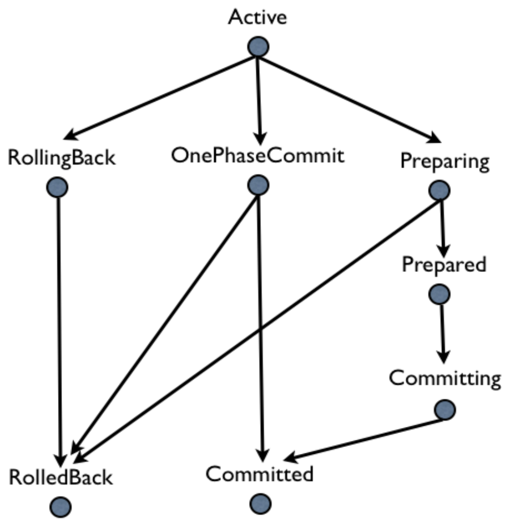

Transaction Manager (factory)
2: Created + transaction link
1: Create
Transaction coordinator
5: Terminate
4: Enlist
6: 2PC
Client
Participant
Service
3: Invocation + transaction link

These components are enumerated below and discussed in the following sections:

- Transaction Manager: this is a factory resource that is responsible for creating new transactions. Once created, the transaction manager has no further role to play in the life of the transaction.
- Transaction Coordinator: this is a specific resource for the transaction. It drives the two-phase commit protocol and manages interactions with participants.
- Client: the user of transactions.
- Service: a transaction-aware service that performs work that may need to be coordinated with other such services elsewhere.
- Participant: a resource that manages the state changes performed by the service in the context of a transaction. The participant is driven through two-phase commit by the coordinator.

4.3 STATE TRANSITIONS

A transaction coordinator and two-phase participant go through the state transitions shown:



As such, all of the resources in the protocol have statuses that can be represented as one of these values. Asking a resource to change its state from, say, Active to Committed, may drive it through all of the intermediate states and as a result trigger protocol specific events, such as driving the two-phase commit protocol.

There are new media types to represent the state of a coordinator and its participants, e.g., application/txstatus, which represents a return type based on the scheme maintained at www.rest-star.org. For example:

tx-status=TransactionActive

Other media types, e.g., tx+xml, are used to represent additional information, such as additional status information, including the time the transaction was created, the number of participants within the transaction etc [9].

Understanding state and how it relates to transactions has influenced our approach to the REST transaction protocol. We have tried to ensure that the protocol embraces HATEOAS principles rather than just using HTTP as a means of conveying message protocols. For instance, if we consider the two-phase commit protocol, one way of instructing a participant to prepare and commit would be through the use of multiple URIs, such as /participant/prepare and /participant/commit, where the root of the URI (/participant) is the actual participant resource on which the protocol is ultimately operating and whose state is ultimately being changed as a result. A POST request on these URIs could then be used to trigger the relevant operation.

However, as you will see in the remainder of this paper, we took a different approach; one which is intimately tied to state management and which we believe is more in the HATEOAS approach. Rather than define a URI per operation, our protocol requires a single URI for each participant (as well as coordinator) and the invoker (e.g., the coordinator) requests that the participant change its state to the relevant value via PUT, e.g., to prepare a participant the coordinator would PUT the status Prepare to the URI.

In the next sections we shall break down the protocol into its various actors and the ways in which they can interact.

4.4 CLIENT INTERACTIONS

In the REST-AT protocol the transaction manager (*transaction factory*) is represented by a resource. In the rest of this specification we shall assume it is *http://www.fabrikam.com/transaction-factory*, but it could be any URI, since no protocol information is inferred by the structure.

In the next few sections we shall describe the various ways in which the client interacts with the transaction factory and the other components illustrated in the previous architecture diagram. Although uniform URI structures are used in the examples that follow, they can be of arbitrary format and the protocol does not require a specific structure for the URIs.

4.4.1 *Creating a transaction*

In order to use a transaction it must first be created, or begun. A client accomplishes this by issuing a POST request to the transaction factory resource. A successful invocation will return the Location header with the URI of the newly created transaction-coordinator resource, which is only specific to the newly created transaction. Several additional related URIs are also returned, as illustrated in the example response from the transaction factory:

HTTP 1.1 201 Created

Location: /transaction-coordinator/1234

Link:</transaction-coordinator/1234/terminator>; rel="terminator",

Link:</transaction-coordinator/1234/participant>; rel="durable",

Link:</transaction-coordinator/1234/anotherparticipant>; rel="volatile"

One of the Links (the *terminator*) is used to end the transaction. The other two Links (the *enlisting* resources) are used for participating in the transaction. The first (durable) is mandatory and is for the traditional two-phase commit protocol, whereas the second (volatile) is optional and is only returned if the implementation supports the volatile two-phase commit protocol, which we describe in a later section.

Note that performing a HEAD on the coordinator must return the same link information.

Once the transaction is created it will normally be terminated by the client issuing a commit or rollback request. However, in case of failures such as a client crash, a timeout is associated with every transaction. If the transaction has not been terminated explicitly before the timeout elapses, the system will automatically roll it back. Performing a POST as shown below will start a new transaction with the specified timeout in milliseconds.

POST /transaction-factory HTTP/1.1

timeout=1000

If the transaction is rolled back 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 will return HTTP response code 410 if the implementation records information about transactions that have rolled back, (not necessary for presumed rollback semantics) but at a minimum must return response code 404, i.e., that the specified resource cannot be found. The invoker can assume this was a rollback.

Performing a GET on the transaction factory returns a list of all transaction coordinator URIs, both active and those performing recovery due to failures. Other statistical information, such as the number of transactions that have committed and aborted, may also be returned by an implementation.

```
<transaction-factory>
  <active-transactions>2</active-transactions>
  <coordinator>/transaction-coordinator/1234</coordinator>
  <coordinator>/transaction-coordinator/5678</coordinator>
  <committed>4567</committed>
  <rolledback>72</rolledback>
</transaction-factory>
```

In the current protocol, attempting to DELETE any transaction will return a 403. However, we are currently discussing whether or not it would be beneficial to map DELETE to a rollback request.

4.4.2 *Obtaining the transaction status*

Performing a GET on a specific transaction, e.g., /transaction-coordinator/1234, returns its current status if the appropriate media type is specified:

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

tx-status=Active

It is possible for a client to try to obtain other information, such as that related to any registered participants, if it specifies the *application/txstatusext+xml* media type in the GET request. An implementation may choose to require authenticated credentials from the client in order for this to be successful.

4.4.3 *Terminating a transaction*

In order to commit or roll back the transaction, the client can PUT the relevant status change to the terminator resource. For example, 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 either return HTTP code 410 or 404 as discussed earlier.

PUT /transaction-coordinator/1234/terminator HTTP/1.1

Content-Type: application/txstatus

Content-Length: --

tx-status=Commit

The state of the transaction must be Active for this operation to succeed. If the transaction is in an invalid state then the implementation will return the HTTP 403 code. Otherwise it may return either 200 or 202 HTTP codes (OK and Accepted, respectively). In the latter case the Location header should contain a URI upon which a GET may be performed to obtain the asynchronous transaction outcome later.

4.5 TRANSACTION CONTEXT PROPAGATION

When making an invocation on a resource that needs to participate in a transaction, either the coordinator URI or the enlisting URI (e.g., /transaction-coordinator/1234/participant) needs to be propagated to the resource. The following approaches are recommended.

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

4.6 COORDINATOR AND PARTICIPANT INTERACTIONS

A two-phase aware participant is registered with the coordinator by POST-ing on the enlisting resource obtained from the transaction factory when the transaction was created originally:

POST /transaction-coordinator/1234/participant HTTP/1.1

participant=/participant-resource/+

terminator=/participant-resource/terminator

Here the terminator resource is the entity with which the coordinator will interact and drive through the two-phase state changes we mentioned earlier.

A successful POST will return HTTP code 201 and typically also the Location header will point to a URI that the participant may use later for recovery purposes.

HTTP/1.1 201 Created

Location: /participant-recovery/1234

In many cases, clients will need to interact with services and resources that are not transaction aware. These services may still provide logical mechanisms that are similar to prepare, commit, and rollback. The specification defines a protocol and media type for registering tx-unaware participants. Due to space constraints we do not provide the details for this protocol.

Performing a GET on the participant resource will return the current status.

4.6.1 Terminating a participant

The coordinator drives the participant through the two-phase commit protocol by a PUT request to the participant's terminator URI with the relevant state change as the message content:

PUT /participant-resource HTTP/1.1

Content-Type: application/txstatus

Content-Length: --

tx-status=Prepare

If the operation fails, e.g., because a participant cannot be committed, then the protocol requires that implementations return the 409 code, i.e., Conflict. Furthermore, depending upon the point in the two-phase commit protocol where such a failure occurs, the transaction must be rolled back to ensure consistency. If the transaction coordinator receives any response other than 200 for Prepare then the transaction will rollback automatically.

If the participant is not in the correct state for the requested operation, e.g., it receives a Prepare when it has been already been prepared, then HTTP gives us another convenient code to return: 412, i.e., Precondition Failed.

The protocol allows the read-only optimization of two-phase commit that we mentioned earlier, to be modeled as a DELETE request from the participant to the coordinator. In this way, the participant can remove itself from the coordinator after prepare is called and no further invocations will occur.

4.7 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).

However, sometimes it is possible that a participant may crash and recover on a different URI, e.g., the original machine is unavailable. In such a situation it may be that the transaction coordinator is unable to complete the transaction, even during recovery, because it cannot contact a recovered participant.

As a result the REST-AT protocol provides a way for a recovering participant to update the information maintained by the coordinator on its behalf. Performing a PUT on the /participant-recovery URI returned by the coordinator during the initial enlistment will overwrite the old participant URI with the new one supplied, telling the coordinator where the participant is now located.

PUT /participant-recovery/1234 HTTP/1.1

new-address=URI

4.8 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 [5] 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 protocol is accomplished in this specification by supporting an additional two-phase commit protocol that encloses the two-phase protocol we have already discussed. This protocol 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 and the coordinator will not record any durable information on behalf of such participants.

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 may not be informed about the transaction outcome. If that is a necessity then they should register with the Durable protocol instead.

5. IMPLEMENTATION

We have implemented a prototype [10] of the protocol in Java. The choice of Java and HTTP naturally led us to use the JAX-RS API [11], which is part of the Java Enterprise Edition standard. JAX-RS is the Java language support for building REST based applications - it is both an annotation-based API for defining resources and a run-time for mapping HTTP requests to Java methods. We use the JAX-RS compliant Resteasy [12] project in our implementation.

Although the algorithm (together with its optimizations) for reaching consensus as described earlier in this paper is simply stated, there are many subtleties and various failure scenarios that must be handled in order to ensure that the protocol guarantees are maintained. Given our industrial credentials, it has been important that we provide a production ready implementation. Therefore we used a mature transaction implementation to implement the atomic guarantees required by the protocol. The JBoss transaction manager (JBossTS) [13] is particularly suitable since it has an API that generates notifications whenever there is a state transition during the execution of the protocol and an API to the transaction log, which is used to ensure persistent state changes can be recorded and replayed reliably during recovery. Because of this flexibility the same core transaction management implementation has been used to implement a range of transaction protocols over the years, including Web Services transactions.

From a design standpoint we created resources to match those architectural components described in Section 4.2. However, the protocol user must implement the participant resources since they require semantic information that is only available to the user. Participant responsibilities are to ensure that changes to a resource can be driven through the consensus protocol, that changes to resources are recoverable in the presence of failures and that changes are durable and isolated from other changes.

The client is responsible for starting and stopping transactions and for propagating the 'transaction URI' during interactions with participants.

5.1 TRANSACTION CREATION

Following the protocol description in Section 4.4, the client sends an HTTP POST request to the transaction factory resource, which in turn uses JBossTS to start a new transaction. A transaction coordinator resource is created to represent the transaction and its URL is returned to the client in the HTTP response Location Header.

Any language that provides an HTTP API can be used to implement the client. For example a Java based client might look like the code snippet shown in Listing 1.

```
import java.net.HttpURLConncetion;
...
// the well-known URL for creating transaction resources
String TXN_MGR_ADDR =
“http://127.0.0.1:8080/rest-tx/tx/transaction-manager”;

HttpURLConnection connection = new URL(TXN_MGR_ADDR).
    .openConnection();
connection.setRequestMethod(“POST”);
connection.setDoOutput(true);
OutputStream os = connection.getOutputStream();

os.write(new String("timeout=1000").getBytes());
os.flush();

/*
 * check that the transaction coordinator
 * resource was created:
 */
if (connection.getResponseCode() != HTTP_CREATED)
    // something went wrong

/*
 * the Location header of the response contains
 * the transaction URL
 */
String transactionURL =
    connection.getHeaderField("Location”);
```

Listing 1: Starting a Transaction

Although not shown in the code listing, the client can subsequently examine the status of the transaction by performing a GET request on the location URL which should return a message with Content-Type “application/txstatus” and body “tx-status=Active”

Also returned in the HTTP headers are the *terminator* and *enlisting* resource URIs that the client can parse and propagate to services:

```
Collection<String> linkHeaders;
String participantEnlistmentURL;
String terminatorURL;

linkHeaders = connection.getHeaderFields().get("Link");
participantEnlistmentURL = getLinkHeader(
    linkHeaders, "durable");
terminatorURL = getLinkHeader(linkHeaders, "terminator");

/*
 * the client can pass the participantEnlistmentURL
 * to resources in any way it sees fit. For example,
 * it could pass it using an HTTP query parameter.
 * If pURL is the URL of the participant then the
 * following would suffice:
 */
String query = String.format(“durable=%s",
    URLEncoder.encode(participantEnlistmentURL,
    “UTF-8”));
URLConnection connection = new URL(
    pURL + "?" + query).openConnection();

doStuffWithParticipant(connection);
doStuffWithOtherParticipants(query);
```

Listing 2: Propagating the Transaction

Later when the client is finished interacting with other participants it will force all the changes to be made durable by ending the transaction (by performing an HTTP PUT request to the terminatorURL). The URL for creating a transaction is mapped to a method on the transaction factory using JAX-RS annotations. When the coordinator implementation starts a transaction it creates an instance of the JBossTS Java transaction class and invokes the begin method.

5.2 TRANSACTION ENLISTMENT (PARTICIPANT PERSPECTIVE)

The client interacts with a participant and passes either the '*terminator*' or '*enlisting*' URL along with HTTP requests. In the previous listing we showed the client propagating the URL using an HTTP query parameter (though using a Link header is probably preferable).

The participant should now associate a unit of work with the transaction by creating '*participant*' and '*terminator*' resources which are passed to the coordinator resource as Link headers by sending an HTTP POST request to the '*enlisting*' URL (which it received from the client). The response to the POST contains a URL in the Location header that the participant should durably record for recovery purposes (for example if it fails and then migrates to a new server then it will use the URL to inform the coordinator that it has moved, as we discussed in Section 4.7).

5.3 TRANSACTION ENLISTMENT (COORDINATOR PERSPECTIVE)

On receipt of the POST request via the '*enlisting*' URL, the coordinator resource creates a representation of the participant that it will use to drive the participant through the two-phase commit and/or recovery protocols later. Within JBossTS there is a helper class, AbstractRecord, which can be used to represent a range of transactional participants from databases through to file systems. We create a specific instance of this class to represent the REST-AT participant. This record is then enlisted with the transaction. When the coordinator subsequently executes the termination or recovery protocols the AbstractRecord instance will be notified.

Since AbstractRecords also hook into the recovery sub system, they are also notified when durable state must be read or written (restore_state and save_state). An important piece of information that is saved is the '*recovery*' URL that the coordinator passed to the participant during enlistment. The URL is used by the coordinator to discover whether a participant has moved during transaction completion.

5.4 TRANSACTION COMPLETION

When the client commits or cancels the transaction, it sends an HTTP PUT request to the transaction '*terminator*' URL. The implementation of the resource representing '*terminator*' locates the instance it created to represent the transaction and calls the appropriate commit operation on it. If the coordinator has failed and restarted then the transaction will not be found and the client can infer "presumed rollback" semantics.

When commit is called, JBossTS calls prepare on the participant records enlisted with the transaction (unless the one-phase optimization applies), and the participant record in turn issues an HTTP PUT request to the real participant resource (via the participant '*terminator*' URL). The transaction system then calls commit or rollback on each participant record depending on the results of the prepare phase.

The commit request from the client is mapped onto a method in the coordinator (using JAX-RS annotations) as show in the next code listing:

```
@javax.ws.rs.PUT
@javax.ws.rs.Path("transaction-manager/{TxId}/terminate")
public Response endTransaction(
    @PathParam("TxId")String txId, String content)
```

```
{
    Transaction tx = lookupTransaction(txId);
    if (tx == null)
        // then infer "presumed rollback" semantics

    /*
     * the content contains content type
     * application/txstatus which indicates whether the
     * client is commiting or rolling back the txn
     */
    boolean commit = isCommit(content);
    /*
     * tell JbossTS to associate the transaction
     * with the current thread
     */
    AtomicAction.resume(tx);

    // and commit or abort it
    commit ? tx.commit() : tx.abort();

    AtomicAction.suspend();
    ...
}
```

When tx.commit() is invoked the transaction system calls into the AbstractRecord mechanism discussed in the previous section and calls the topLevelPrepare() and topLevelCommit() methods of the participant record.

5.5 RECOVERY

There is little benefit in providing transactional integrity unless recovery is properly addressed. If the coordinator fails after it has reached its commit decision then it will have logged that decision and the recovery sub system takes over the responsibly for completing the transaction. Similarly if a participant fails after the commit decision is logged the recovery sub system will periodically retry the final phase of the consensus protocol.

The recovery system (which runs separately from the transaction factory) scans a transaction log looking for outstanding records that have initially failed to complete the second phase of the consensus protocol and attempts to replay that phase on the record. The log contains durable representations of participant records. So, in effect, the recovery system recreates the participant record from the entry in the log and invokes its commit method, which is then able to call PUT on the original participant '*terminator*' URI.

6. CONCLUSIONS AND FUTURE WORK

There is an ongoing discussion about the validity of many enterprise middleware capabilities, such as transactions, to the REST area. Our work on the REST-AT protocol is not an attempt to suggest that REST, or the world of HTTP, needs transactions and without them is failing to deliver on enterprise capabilities. However, in our experience there are a class of applications and use cases where transaction protocols such as REST-AT could simplify their development.

We believe that the protocol outlined in this paper is a good REST citizen. However, the use of transactions within a REST application can break those principles due to the traditional ACID semantics. Fortunately we also believe that there is a solution in what are commonly referred to as *extended transactions*. Traditional transaction processing systems are sufficient to meet requirements if an application function can be represented as a single transaction. However, this is frequently not the case. Transactions are most suitably viewed as short-lived entities, performing stable state changes to the system; they are less well suited for structuring long-lived application functions that run for minutes, hours, days, or longer. Long-lived transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources (usually by locking) for a long time. Furthermore, if such a transaction aborts much valuable work already performed will be undone [14][15].

Many business-to-business applications benefit from transactional support in order to guarantee consistent outcome and correct execution. These applications often involve long running computations, loosely coupled systems and components that do not share data, location, or administration and it is difficult to incorporate atomic transactions within such architectures. Fortunately much work has been done in the area of what are often referred to as extended transactions, which loosen the various ACID semantics [16][17][18][19].

A popular approach is Sagas [17], where services are requested to do work but not in a provisional manner as they are in a traditional transactional setting: the work is done immediately. If the Saga needs to undo (roll back) then it instructs the services to perform some compensation work. This means that in a loosely coupled environment there is no retaining of locks or provisional state change for long durations. Furthermore, in some models there is also no requirement for a centralized coordinator, with information about the transaction being distributed across participants. It is for this reason that the Web Services transaction standard [2] supports an extended transaction model as well as a traditional ACID model, i.e., so that loosely coupled applications that require some aspects of transaction semantics can obtain them without forcing them to become closely coupled.

For similar reasons we have been working on a compensation transaction model for REST [20]. At this stage the compensation protocol is still under development but the goal is to provide something that is not only a good REST citizen but also does not turn a RESTful application that uses it into one that cannot claim to be RESTful. We hope to be able to discuss this protocol in the future as well as demonstrate it with another implementation based on JBossTS.

7. REFERENCES

[1] X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document Number XO/CAE/91/300.

[2] Web Services Transaction Technical Committee, OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx

[3] M. C. Little et al, "Constructing Reliable Web Applications using Atomic Actions", Proceedings of the 6th World Wide Web Conference, April 1997, Santa Clara, CA.

[4] InfoQ discussion, July 2009, http://www.infoq.com/news/2009/06/rest-ts

[5] OMG Object Transaction Service, http://www.omg.org/cgi-bin/apps/do_doc?formal/02-08-07.pdf

[6] "Shootout at the transaction corral; BTP versus WS-T", http://www.objectwatch.com/issue_41.htm

[7] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann, September 1992.

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

[9] REST-AT specification, July 2010, http://www.jboss.org/reststar.

[10] Implementation of the REST-AT protocol: http://anonsvn.jboss.org/repos/labs/labs/jbosstm/trunk/rest-tx/

[11] JSR 311: JAX-RS: The Java API for RESTful Web Services, http://jcp.org/en/jsr/detail?id=311

[12] Resteasy Project: a certified implementation of the JAX-RS specification, http://www.jboss.org/resteasy

[13] JBoss Transactions, http://www.jboss.org/jbosstm

[14] D. J. Taylor, "How big can an atomic action be?", Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp. 121-124.

[15] A. K. Elmagarmid (ed), "Transaction models for advanced database applications", Morgan Kaufmann, 1992.

[16] C. T. Davies, "Data processing spheres of control", IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[17] H. Garcia-Molina and K. Salem, "Sagas", Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.

[18] I. Houston, M. Little, et al. "The CORBA Activity Service Framework for Supporting Extended Transactions", Proceedings of Middleware 2001, Heidelberg, 2001.

[19] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Gunthor and C. Mohan, "Advanced transaction models in workflow contexts", Proc. of 12th Intl. Conf. on Data Engineering, New Orleans, March 1996.

[20] REST Compensation Transaction protocol specification, July 2010, http://community.jboss.org/wiki/CompensatingRESTfulTransactions