2009 IEEE International Conference on Web Services

RETRO: A Consistent and Recoverable RESTful Transaction Model

Alexandros Marinos Amir Razavi Sotiris Moschoyiannis Paul Krause

Department of Computing, FEPS, University of Surrey,

{a.marinos, a.razavi, s.moschoyiannis, p.krause}@surrey.ac.uk

Abstract

With REST becoming a popular paradigm for web services, more and more use cases are applied to it, including some that require transactional guarantees. We propose a RESTful transaction model that satisfies both the constraints of transactions as well as those of the REST architectural style. We provide formal proof of consistency and recoverability in the proposed framework and show the robustness of its properties in the presence of concurrent transactions.

1. Introduction

Representational State Transfer (REST) is an architectural style introduced by Roy Fielding in [7] as a refinement of the architectural style that had emerged in the World Wide Web. The main features of REST include focusing on resources identified by names, a uniform interface to manipulate those resources, hypermedia as a means of linking the resources and statelessness in the client-server interactions.

REST, especially over the HTTP protocol [8], has long been championed as a competing web service paradigm to the WS-* stack. This claim has recently been reinforced with the publication of works such as [17], the more recent [11], together with the recognition of the apparent complexity and lack of adoption of WS-* technologies beyond the corporate firewall [19].

As is common with disruptive technologies, REST over HTTP is evolving to compete with WS-* in increasingly advanced usage scenarios [4], [12]. This paper aims to be part of the next wave of REST evolution by defining a RESTful transaction model that is designed to operate over HTTP. To date, usage of REST has remained at the level of serial sequences of operations, each succeeding or failing atomically. While its advantages have made it a popular web services paradigm on the web, the WS-* stack provides the only standard for unplanned transactions. Web applications have to resort to ad-hoc solutions of variable quality in order to address this need. A prominent example is the shopping cart found in many e-commerce web sites where the items (resources) in a shopping cart have to be purchased in one step and also potentially become unavailable when a customer adds them to a shopping cart, the transactional concepts of atomicity and locking respectively. Also, there is no reason not to consider traditional service composition scenarios such as a travel scenario necessitating the booking of a flight, hotel and rental car, executed over the web-facing APIs of different providers, something currently not possible.

Transactions have been defined in terms of the four properties contained in the ACID acronym [9]. These properties guarantee that a system is maintained in a consistent state, even as transactions are executed within it concurrently. This includes the situations where one or more transactions fail to commit. When dealing with a sequence of transactions (one transaction executed at a time), each transaction starts with the consistent state that its predecessor ended with. If all the transactions are short, the data are centralised in a main memory, and all data are accessed through a single thread, then there is no need for concurrency. The transactions can simply be run in sequence. Real-world interactive systems however, often require the execution of several transactions concurrently. Use cases such as distributed environments [4,22] or dynamic allocation of resources to external developers [21] illustrate this need.

While transactions are concerned with the constraints of adhering to ACID properties, REST adheres to its own set of constraints. These are primarily expressed by the uniform interface constraint, but supported by the following four constraints: Resource Identification, Resource manipulation through representations, Self-descriptive messages, and Hypermedia as the engine of application state [20]. Our efforts are directed at creating a truly RESTful transaction model that satisfies both the constraints of REST and the ACID constraints relevant to transactions. In this paper we describe a RESTful framework for transactions (RETRO) in which the locking scheme necessary for ACID transactions is adapted to work within the architectural style of REST.

We provide a more rigorous justification of the need for our new model in the next two sections. Following that, we discuss how locks may be introduced for concurrency control into RESTful working over HTTP. In Section 5, we then elaborate this into a two-phase lock model and demonstrate that the result is consistent, wormhole free, and supports recoverability. Proofs of soundness and completeness of the resulting model are provided in Section 6, and then conclude.

2. Relevant work

Various approaches have been proposed to support concurrent execution of transactions, but locking has

978-0-7695-3709-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICWS.2009.99

IEEE computer society

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

emerged as the most feasible solution [2], [15], [9], [16]. Additionally, [3] and [14] use similar principles with different semaphores. When implementing a lock mechanism, it is important to ensure that concurrent execution does not have lower throughput or much higher response times than serial execution. The second major concern is to avoid high computational overhead (see concurrency control laws in [9]).

The application of the transactional concept in WS-* adds considerable complexity to the required coordination framework [21]. This can be seen more clearly when analysing the pattern behaviour for the recovery model (*compensation*) [22], [23]. In contrast, REST works directly with resources. This is in line with the semantics of the basic theorems in conventional transaction processing [9]. Transactions rely on read/write operations on objects and RESTful HTTP, likewise, provides GET (equivalent to 'read') and DELETE, POST, PUT (equivalents of 'write') methods.

Various approaches have been proposed for handling RESTful transactions. The traditional approach is to simply design a new resource that can be used to trigger the desired transaction on the server side. For example, to transfer funds from one bank account to another, there could be a 'transfer request' resource to which new 'transfer requests' can be posted. While it can be very simple to implement at design time, this constrains users to the predictive ability of the developers. Also, in scenarios where a large or unpredictable variation of transactions may take place, all the necessary resources cannot have been designed beforehand. This situation is similar to the static versus dynamic allocation debate found in the database and transaction literature [2], [9]. The approach completely breaks down however, when a transaction exceeds the scope of a single provider, the case of distributed transactions. Other approaches such as [13] suggest extending REST to include mutex locks, but this would necessitate extending HTTP as well.

The alternative to these approaches is to introduce locks on resources by modeling them as resources themselves [17]. While this approach looks much more capable, the details of its implementation and its extension into transactions have neither been fleshed out nor proven. In this paper we describe how this approach can be extended to produce a fully specified and theoretically robust RESTful transaction model.

3. Concurrency issues in RESTful HTTP

The classic view taken in addressing the *isolation* property is to consider transactions in terms of inputs and outputs [9], [6]. These are essentially *read* (input) and *write* (output) operations. Write operations are described as operations that affect the state of resources. On the other hand, REST prescribes a uniform interface for accessing resources. One challenge is therefore to map the traditional input/output perspective with the RESTful approach to the uniform interface. Since our model operates over the HTTP protocol, we examine its four resource interaction operations.

GET is the standard retrieve operation. Its execution must be safe; it should have no side-effects. It should also be idempotent. Duplicate messages should have no adverse effects. POST is understood as an operation to create a new resource on a server where the target URI is not known. The representation of the resource is sent via POST to the collection that will contain the resource. The server determines its appropriate location and the resulting URI is returned to the client as part of the response. POST is neither safe nor idempotent. PUT can be used for updating resources, by simply instructing the server to apply a new representation as a replacement of the previous one. It can also be used to create a new resource, when a representation is PUT at a URI that was previously unused. A very important point is that a PUT operation may correspond to a Create or an Update operation in the CRUD paradigm, and sometimes the client may not even know which of the two is going to be applied. This depends solely on the state of the server. Finally, DELETE is used to request removal of the resource representation at the target URI.

All the operations described above are used to manage the lifecycle of the resources directly related to the transaction itself. However, the transactions our model can orchestrate are only those that intend to perform GET and PUT operations. In the case of PUT, since we guarantee that the resource exists before it is PUT to, we are only dealing with the 'update' capacity of the operation and not its 'create' aspect. In this sense, the only type of non-safe operation ('write') that our model currently supports is PUT, in its update capacity. Within the scope of these assumptions, the term 'PUT' is used as equivalent to 'write' for the rest of this paper.

As GET operations do not change the state of resources, provided the initial state of a resource is consistent, concurrent GET requests to the same resource cannot cause inconsistency. On the contrary, PUT operations of different transactions on the same resource change the state of the resource and may violate consistency or isolation. While we can assume that a transaction "knows what it is doing" in terms of its internal data manipulation, overlap between PUTs of one transaction and GET actions of another, can violate isolation and cause inconsistency.

Additionally, PUT-related interactions between different concurrent transactions on the same resource can also cause a problem. If we consider GET operations as inputs of transactions and PUTs operations as output operations of them, this can be expressed as:

$$\textbf{EQ. 1:} \qquad O_i \cap (I_j \cup O_j) = \emptyset \ \text{for all} \ i \neq j$$

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

Where I_j denotes the set of resources accessed via GET by transaction T_k (its inputs), and O_j the set of resources altered via PUT by transaction T_j (its outputs). Based on EQ.1, it is appropriate to say that the set of transactions $\{T_i\}$, whose outputs are disjoint from one another's inputs and outputs, can run in parallel with no concurrency anomalies.

We define '*history*' as any sequence-preserving merge of the actions of a set of transactions into a single sequence .A history is denoted $by\ H = \langle\langle t, a, r\rangle_i | i = 1, \dots, n\rangle$. Each *step* of the history is a tuple $\langle t, a, r\rangle$ comprising an action *a* by transaction *t* on resource *r*. A history for the set of transactions $\{T_j\}$ is a sequence, containing each transaction T_j as a subsequence and containing nothing else. Essentially, a history lists the order in which actions were successfully completed.

Serial histories are one-transaction-at-a-time histories. Since no concurrency is induced in serial histories, there is no interdependency between transactions. Therefore wormholes or inconsistencies will not be an issue. While this is a useful theoretical aspect, in reality transactions can have any order and hence histories will not be serial.

3.1. Concurrency anomalies

In this section we will analyse the result of executing transactions concurrently, in a RESTful manner, and highlight the potential concurrency anomalies that arise.

When two (or more) transactions access the same resource, they may produce two (or more) different versions of that resource (*lost update*), or simply they may work with the out-of-date version of the resource (*dirty* GET and *unrepeatable* GET). Fig. 1 shows these three inconsistent scenarios.

As shown in Fig 1, interleaved RESTful interactions by multiple parties may cause several concurrency issues. A transaction GETs a resource twice, once before another transaction's PUT action and the second one after the PUT action (the second transaction may PUT a new version and commit). This means a transaction changes the resource (PUT), when another transaction had ongoing access (GET) to it and has not finalised its access. On the other hand, the first transaction has to deal with inconsistent GETs on the same resource.

The second classical problem is '*Lost updates*' and it occurs when the first transaction's PUT is overwritten by the second transaction which uses PUT based on the initial value of the resource (second scenario in Fig. 1). This means one of the updates will be overwritten without being taken into account.

Finally, a problem can also occur when a transaction relies on out-of date resources (Fig. 1). A transaction GETs a resource between two PUT operations by another transaction. As a result, the transaction may use an inconsistent resource state as the other transaction has not finished its updates on the resource and may even roll back, rendering the retrieved representation invalid.

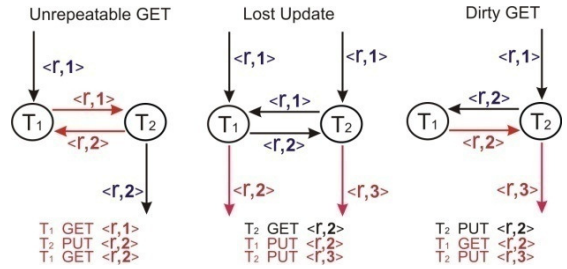

Figure 1 – Concurrency challenges

.

Fig 1 shows the simplest scenarios of these problems, but they may be easily extended to multi transactions where accessing resources are a sequence where it comes back to the first transaction. On the other hand, accessing a resource may look like a cycle when we try to draw a sequence diagram for them. These classical transactional problems are called *wormholes.* In the next section, we try to provide a clear definition for them in terms of RESTful transactions.

3.2 Wormholes

We start by defining dependencies between transactions in a history. A transaction *T* is said to be dependent on another transaction *T'* in a history *H* if *T* GET (reads) or PUT (writes) data-resources previously PUT (written) by *T'* in the history *H*, or if *T* PUT (writes) a resource previously GET (read) by *T'*.

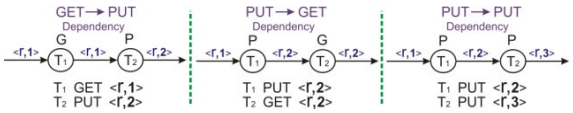

Figure 2 – Types of dependencies

We can formalise different types of dependencies (shown in Fig. 2) through a *Dependency Graph* where nodes are 'transactions', arcs indicate 'transaction dependencies', and labels on arcs denote 'resource versions'. The *version* of a resource *r* at step *k* of a history is an integer and is denoted by *V(r,k)*. In the beginning each resource has version zero (*V(r,0)*=0). At step *k* of a history *H*, resource *r* has a version equal to the number of writes to that resource before this step. This is put formally as follows.

$$V(r,k) = \left|\{\langle t_j, a_j, r_j\rangle \in H \mid j < k \text{ and } a_j = \text{PUT and } r_j = r\}\right|$$

The outer vertical bars represent the set cardinality function. Each history, *H*, for a set of transactions $\{T_i\}$ defines a ternary *dependency relation* DEP(*H*), defined as follows. Let *T*1 and *T*2 be any two distinct transactions, let *r* be any resource, and let *i*, *j* be any two steps of *H* with $i < j$. Suppose step $H[i]$ involves action *a*1 of *T*1 on resource *r*, step $H[j]$ involves *a*2 of *T*2 on *r*,

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

and suppose there is no PUT on r by any transaction between these steps (there is no $\langle T', PUT, r\rangle$ in $H[i+1], \ldots, H[j-1]$). Then DEP(H) is defined as:

$\langle T, \langle r, V(r,j)\rangle, T'\rangle \in \text{DEP}(H)$
if $a1$ is a PUT and $a2$ is a PUT
$a1$ is a PUT and $a2$ is a GET
$a1$ is a GET and $a2$ is a PUT.

PUT→PUT, PUT→GET and GET→PUT dependencies.

The dependency relation for a history defines a directed *dependency graph*, where transactions are the nodes of the graph, and resource versions are label on the edges. If $\langle T, \langle r, j\rangle, T'\rangle \in \text{DEP}(H)$, then the graph has an edge from node T to node T' labeled by $\langle r, j\rangle$. Two histories are equivalent, if they have the same dependency relation.

The dependency relation of a history defines a time order of the transactions. Conventionally this ordering is signified by <<< and it is the *transitive closure* of $<<<_{\text{H}}$. It is the smallest relation satisfying the equation $\text{T} <<<_{\text{H}} \text{T}'$ if $\langle \text{T}, \text{r}, \text{T}'\rangle \in \text{DEP}(H)$ for some resource version r, or $\text{T} <<<_{\text{H}} \text{T}''$ and $\langle \text{T}'', \text{r}, \text{T}'\rangle \in \text{DEP}(H)$ for some transaction T″ and some resource r. Whenever $T <<< T'$ there is a path in the corresponding dependency graph from transaction T to transaction T'. The <<< ordering defines the set of all transactions that run before or after T as follows.

$$\text{BEFORE}(T) = \{T' | T' <<< T\}$$
$$\text{AFTER}(T) = \{T' | T <<< T'\}$$

If T runs fully isolated (ex: it is the only transaction, or it GET and PUT resources not accessed by any other transactions), then its BEFORE and AFTER sets are empty (it can be scheduled in any way). When a transaction is both after and before the other distinct transaction, it is called *wormhole transaction* (T' here):

$$T' \in \text{BEFORE}(T) \cap \text{AFTER}(T)$$

for some resource version r, or (for some transaction , and some resource r).This means that any cycle in a dependency graph is a wormhole. Using a well-formed and two phase locking mechanism is a conventional method for avoiding wormholes [9]. In the next section we describe how such a locking mechanism is adapted to RESTful transactions as a practical way for avoiding wormholes and then prove that our RESTful transaction model is wormhole-free.

4. Locks in RESTful HTTP

In order to handle concurrency challenges in HTTP, we introduce the concept of locks. This is done in a way that does not affect the always available and backwards compatible nature of the web.

4.1. Locking resources

For an API to be characterized as RESTful according to the hypermedia constraint, it must allow a client to interact with the service solely by being given a single URI and understanding of the relevant media types. This enforces loose-coupling and elimination of assumptions.

Lockable Resource (R): Ideally, any resource that can be served by an HTTP server should be lockable regardless of serialization format. This however would require the HTTP protocol to carry the metadata for the locking mechanism. Since we wish to preserve the HTTP protocol, we opt for a fragment of XML that is to be included in an XML representation of a resource. This approach could potentially be extended to other formats such as JSON [5] but not to binary files such as images or zip archives. The information that should be in the fragment is the location of the lock collection and the location of the transaction collection. The inclusion of this fragment (Fig. 3) makes any resource lockable. Namespaces could also be utilized to avoid namespace collision but this would limit the approach to serializations that support namespaces.

```
<lockable>
    <link rel="lock_collection" href="http://example.org/resource/locks/" />
    <link rel="transaction_collection" href="http://example.org/transactions/" />
</lockable>
```

Figure 3 – (R) XML Fragment

Lock Resource (R-L): The lock resource is represented by a dedicated media type and should contain the elements in Table 1.

ResourceURI: a link back to the resource that this lock affects.
TransactionURI: a link to the transaction that controls the lock.
Type: "S" or "X" depending on the type of the lock.
PrevLockURI: a link to the previous lock in the lock sequence.
Timestamp: Server's timestamp when the lock was granted.
Duration: Indicates the interval that the lock has been granted for.
ConditionalResourceURI: A link to the representation of the resource that will come into effect once the lock is committed.

Table 1 - Elements of R-L

The type element can take one of two values, X or S, corresponding to the available lock types. X stands for **XLOCK:** eXclussive Lock, and S stands for **SLOCK:** Shared Lock. To place a new lock, the server must authenticate the user as the owner of the transaction that is referenced by the lock. The length of time of effectiveness that is granted to a lock is dependent on the maximum length of time that the server is prepared to grant a guarantee to the client. Once the duration of the lock expires, the lock is aborted. *To avoid violating 2PL, once a lock of a transaction expires, all other locks of the same transaction expire.*

The result of the GET operation does not change until a lock of type X is committed. In this sense, the locks and transactions are transparent to the GET which on

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

commit reacts as if a simple PUT was applied. This was a specific design objective. PUT and DELETE operations return a *'405 Method Not Allowed'* HTTP response for the duration of a lock's effect. GET requests should still return successfully. This behaviour maintains backwards compatibility, with the understanding that if a client requires further guarantees on the future state of the resource, the client should seek to place a lock. In all other cases, the semantics of GET are unaffected, as a GET on a resource does not guarantee that the state will remain unchanged for any period of time.

4.2. Well-formed collections of locks

As expected, a transaction cannot lock a resource that is locked by another transaction. But if two or more transactions want to GET the content of a resource, they are not going to change the resource state. This will therefore not cause any conflict or access to data which has been PUT to a resource by another transaction, but the first transaction has not committed and may change the version of the resource again). Table 2 shows the lock compatibility. The inferred rules constrain the set of allowed histories. Histories that satisfy the locking constraints are called *legal histories*.

		Mode of Preceding Lock	
		Share	*Exclusive*
Mode Of New Lock	*Share*	*Yes*	*No*
	Exclusive	*No*	*No*

Table 2 – Legal lock sequences

Resource Lock Collection (R-Lc): The R-Lc contains locks in sequences that follow the compatibility rules stated in Table 2, rendering the transaction well-formed. The lock collection is represented as an Atom Feed [12]. Since ATOM does not support sequencing entries, we use the 'PrevLockURI' element of the lock resource to create a linked list of locks. The client can retrieve the lock collection via GET to determine if the resource is locked. An empty feed indicates an unlocked resource. New locks can be submitted to the resource collection via the POST method.

5. Two phase locking and recoverability

In the previous section, we described how our model provides a well-formed locking system for GET and PUT. We now show that by adding two-phase locking, the model becomes wormhole-free. We then show how this facilitates recoverability in RETRO and illustrate the key ideas with a simple example.

GET	Returns the resource's collection of locks.
POST	Adding a new lock to the related resource

Table 3 - Available Operations for R-Lc

5.1. Two phase locking is wormhole free

In two-phase locking [10ref?] each transaction can use locking in two phases. In the first phase (*growth*), it can acquire locks for resources (SLOCK or XLOCK) and in the second phase (*shrink*), it releases them. These two phases should not have any overlap. When the transaction starts to UNLOCK a resource, it cannot lock any more resources under any circumstances. So, unlocking resources means that the transaction is either successfully committing or aborting.

We have seen in discussing '*Lock Resource (R-L)*' (Section 4-1), that each transaction in our RESTful transaction model can use two different types of Locks for its resources (SLOCK for GET and XLOCK for PUT). Therefore, in $H = \langle \langle t, a, r \rangle_i | i = 1, \dots, n \rangle$ we consider two extra actions for 'a' : SLOCK; and, XLOCK. Since these locks at some point should be released, we also have UNLOCK as another action for 'a'. Now, we want to show that if all transactions are well-formed and two-phase, any legal history will be isolated (wormhole-free). In what follows, we first show how the additional actions required for the two-phase locking are incorporated in our well-formed RESTful transactions, and then invoke the well-known Wormhole Theorem from conventional transactions [9] to show that our model is wormhole-free.

Suppose H is a legal history of the execution of a set of transactions, each of which is well-formed and two-phase. For each transaction, T, define SHRINK(T) to be the index of the first unlock step of T in history H . Formally:

$\text{SHIRINK}(T) =$
$min(i | H[i] = \langle T, \text{UNLOCK}, r \rangle \text{ for some resource}))$.

Since each transaction T is non-null and well-formed, it must contain an *UNLOCK* step. Thus SHRINK is well defined for each transaction. First we need to show that if there is path in the dependency graph from a transaction T to a transaction T', then the first unlock step of T will happen before that of T'. This is summarised in the following lemma.

Lemma: If $T <<< T', ten \text{ SHRINK}(T) < \text{SHRINK}(T')$.

Suppose $T <<< T'$, then suppose there is a resource r and steps $i < j$ of history H, such that $H[i] = \langle T, a, r \rangle$, $H[j] = \langle T', a', r \rangle$; either action a or action a' is a PUT (this assertion comes directly from the definition of DEP(H) in section 3). Suppose that the action a of T is a PUT. Since T is well-formed, then, step i is covered by T doing an XLOCK on r. Similarly, step j must be covered by T' doing an SLOCK or XLOCK on r. H is a legal history, and these locks would conflict, so there must be a $k1$ and $k2$, such that:

$i < k1 < k2 < j$ and $H[k1] = \langle T, \text{UNLOCK}, r \rangle$ and

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

either $H[k2] = \langle T,\text{SLOCK},r\rangle$ or $H[k2] = \langle T',\text{XLOCK},r\rangle$.

Because T and T' are two-phase, all their LOCK actions must precede their first UNLOCK, action; thus, $\text{SHRINK}(T) \leq k1 < k2 < \text{SHRINK}(T')$. This proves the lemma for the $a = \text{PUT}$ case. The argument for the $a' = \text{PUT}$ case is almost identical. The SLOCK of T will be incompatible with the XLOCK of T'; hence, there must be an intervening $\langle T,\text{UNLOCK},r\rangle$ followed by a $\langle T',\text{XLOCK},r\rangle$ action in H. Therefore, if $T <<< T'$, then $\text{SHRINK}(T) < \text{SHRINK}(T')$. Proving both these cases establishes the lemma. We may now invoke the Wormhole Theorem [9] and infer that H is wormhole-free by contradiction.

Assume that H is not wormhole-free. Then the Wormhole Theorem dictates that there must be a sequence of transactions $\langle T_1, T_2, T_3, \ldots, T_n\rangle$, such that each is before the other (i.e., $T_i <<<_H T_{i+1}$), and the last is before the first (i.e., $T_n <<<_H T_1$). Using the above lemma, this in turn means that $\text{SHRINK}(T_1) < \text{SHRINK}(T_2) < \cdots < \text{SHRINK}(T_n) < \text{SHRINK}(T_1)$. Hence, we have $\text{SHRINK}(T_1) < \text{SHRINK}(T_1)$ which gives the desired contradiction. Thus, H cannot have any wormholes.

5.2. Transaction Resource

Determining the scope of each transaction and whether it is in a GROWTH or SHRINK phase is necessary. We therefore introduce the required resources.

Transaction (T) : This resource can be represented by a dedicated media type (e.g. application/vnd.retro-transaction+xml), containing the elements in Table 4.

TransactionCollectionURI:
OwnerURI:
TransactionLockCollectionURI:

Table 4 - Elements of T

These elements identify the resources vital a transaction. The owner of the transaction can locate these collections by GETting the transaction resource.

Transaction Collection (Tc): The transaction collection is a resource where new transactions are submitted via the POST operation which creates a new transaction and returns the URI for its representation. The resource itself cannot be accessed via GET as the clients that need to know the location of a specific resource are informed at the time of POSTing.

Transaction Lock Collection (T-Lc) : The transaction lock collection contains links to the locks that belong to a specific transaction, formatted as an Atom feed. Clients cannot abort single locks directly but must do so through the T-Lc which aborts all the locks of a transaction, leaving the transaction void and is equivalent to aborting the transaction.

GET	Returns the collection of locks relevant to a transaction
DELETE	Aborts all the locks of the relevant transaction. This can only be performed by an owner of the transaction.

Table 5 - Available Operations for T-Lc

5.3. Recoverability

Based on the Rollback Theorem, a transaction that unlocks an exclusive lock and then performs a '*Rollback*' is not well-formed and can potentially cause a wormhole unless the transaction is degenerated. As the theorem is well-known, we refer the interested reader to [9] for the proof. The important point of the theorem is that we have to degenerate the transaction to effect rollback. For this purpose, our model does not store potential updates on the actual resources but works on the shadow of the locked data, the *conditional resource representation*.

Conditional Resource Representation (R-C): A resource that is of identical media type as the locked resource. The conditional resource representation is essentially the state that will be applied to the resource once the XLOCK is committed.

GET	Returns the representation that will be committed if the relevant XLOCK is committed.
PUT	Creates a new conditional state that will replace the current state of the locked resource once the linking XLOCK is committed.
DELETE	Deletes the conditional state. If the XLOCK is committed, there will be no write action performed.

Table 6 - Available Operations for R-C

5.4. Model overview

Having defined all the resource types, it is easy to see that an interconnected network arises. Figure 4 displays the interconnections of the resource graph. It can be observed that having a URI for R is enough to locate all other resources in the network. The connection from Tc to T is different from the other connections as there is no GET ability for the Tc resource, for security reasons. The URI of a given T is only returned as a response to the initial POST operation on Tc performed by the transaction's owner.

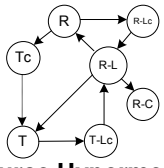

Figure 4 – Resource Hypermedia connections

Table 7 summarizes all the relevant resource types that comprise our model together with a short description and a list of the allowed operations.

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

Client	Operation	Resource	Response	Description
A	GET	R2	200 OK	GETting R2 to extract location of TC and R2-LC
A	POST <new transaction>	TC	201 CREATED {Location: T1}	Creating a new transaction
A	POST <LOCK {type:X}>	R2-LC	201 CREATED {Location: R2-L1}	POSTing an XLOCK to R2-LC
B	GET	R1	200 OK	GETting R1 to extract location of TC and R1-LC
B	POST <new transaction>	TC	201 CREATED {Location: T2}	Creating a new transaction
B	POST <LOCK {type:S}>	R1-LC	201 CREATED {Location: R1-L1}	POSTing an SLOCK to R1-LC
A	GET	R1	200 OK	GETting R1 to extract location of R1-LC
A	POST <LOCK {type:S}>	R1-LC	201 CREATED {Location: R1-L1}	POSTing an SLOCK to R1-LC
B	GET	R1	200 OK	GETting the locked representation of R1
A	GET	R1	200 OK	GETting the locked representation of R1
A	GET	R2	200 OK	GETting the locked representation of R2
B	GET	R2	200 OK	GETting R2 to extract location of R2-LC
B	POST <LOCK {type:X}>	R2-LC	403 Forbidden	POSTing an XLOCK to R2-LC. R2 is locked, POST fails.
A	GET	R2-L1	200 OK	GETting R1 to extract location of R2-L1-CR
A	PUT <new version>	R2-L1-CR	201 CREATED	Creating a conditional Representation of R2
A	DELETE	T1	200 OK	Commiting R2-C to R2 and Unlocking R1 and R2
B	POST <LOCK {type:X}>	R2-LC	201 CREATED {Location: R2-L1}	POSTing an XLOCK to R2-LC
B	GET	R2	200 OK	GETting the locked representation of R2
B	PUT <new version>	R2-C	201 CREATED	Creating a conditional Representation of R2
B	PUT <new version>	R2-C	200 OK	Updating the conditional Representation of R2
B	DELETE	T2	200 OK	Commiting R2-C to R2 and Unlocking R1 and R2

Figure 5 – example of two transactions operating on the same resources

The example in Figure 5 shows how two separate transactions can safely operate on the same resources, purely through HTTP operations. We can also see that while the two transactions are able to place an SLOCK on R1, client B is not allowed to XLOCK R2 while client A already has an XLOCK on it, a direct application of the lock compatibility rules seen in Table 2. Instead, client B continues the transaction when R2 is unlocked.

Lockable Resource (R)	A resource that locks can be applied to Operations: GET, [By XLOCK owner: PUT]
Resource Lock Collection (R-Lc)	The collection of locks that apply to a particular resource. Operations: GET, POST
Lock Resource (R-L)	The representation of a specific lock Operations: GET
Conditional Resource Representation (R-C)	The potential representation of a locked resource, once its lock is committed. Operations: GET, [By XLOCK owner: PUT, DELETE]
Transaction Collection (Tc)	The collection of transactions on the server. Operations: POST
Transaction Resource (T)	The representation of a specific transaction. Operations: GET
Transaction Lock Collection (T-Lc)	The collection of locks connected to a specific transaction. Operations: GET, [By transaction owner: DELETE]

Table 7 – Resources and operations

6. Soundness / Completeness

One may argue the necessity of a well-formed and two-phase history, which our approach carefully follows. To prove the soundness of these properties, we use the converse locking theorem [9]. If a transaction is not well-formed or two-phase, it is possible to write another transaction such that the resulting pair has a legal but not isolated history, unless the transaction is degenerated.

If transaction $T = \langle\langle T, a_i, r_i\rangle | i = 1, \dots, n\rangle$ is not well-formed and not degenerated, then for some k, $T[k]$ is a GET or PUT action that is not covered by a lock. The GET case is proved here; the PUT case is similar.

Let $T[k] = \langle T, \text{GET}, r\rangle$. Define the transaction,

$$T' = \langle\langle T', \text{XLOCK}, r\rangle, \langle T', \text{WRITE}, r\rangle, \langle T', \text{WRITE}, r\rangle, \langle T', \text{UNLOCK}, r\rangle\rangle$$

That is, T' is a double update to resource r. By inspection, T' is two-phase and well-formed. Consider the history;

$$H = \langle T[i] | i < k\rangle \| \langle T'[[1], T'[[2], T[k], T'[3], T'[4]\rangle \| \langle T[i] | i > k\rangle$$

That is, H is the history that places the first update of T' just before the uncovered GET and the second update just after the uncovered GET. H is a legal history, since no conflicting locks are granted on resource r at any point of the history. In addition, for some j, $\langle T', \langle r, j\rangle, T\rangle$ and $\langle T, \langle r, j\rangle, T'\rangle$ must be in the DEP(H); hence, $T <<_{H} T' <<<_{H} T$. Thus T is a wormhole in the history H. Invoking the wormhole theorem, H is not an isolated history. Intuitively, T will see resource r while it is being updated by T'. This is a concurrency anomaly.

Now it is possible to show, if a history is not two-phase it can be legal but not isolated;

Suppose that transaction $T = \langle\langle T, a_i, r_i\rangle | i = 1, \dots, n\rangle$ is not two-phase and not degenerate.

Then for some $j < k$, $T[j] = \langle T, \text{UNLOCK}, r1\rangle$ and $T[k] = \langle T, \text{SLOCK}, r2\rangle$ or $T[k] = \langle T, \text{XLOCK}, r2\rangle$.

Define the transaction

$$T' = \langle \langle T', \text{XLOCK}, r1\rangle, \langle T', \text{XLOCK}, r2\rangle, \langle T', \text{WRITE}, r1\rangle, \langle T', \text{WRITE}, r2\rangle, \langle T', \text{UNLOCK}, r1\rangle, \langle T', \text{UNLOCK}, r2\rangle \rangle$$

That is T' updates resource $r1$ and $r2$. By inspection, T' is two-phase and well-formed. Consider the history:

$$H = \langle T[i] | i \le j\rangle \| T' \| \langle T[i] | i > j\rangle$$

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.

This says that H is the history that places T' just after the UNLOCK of $r1$ by T. H is a legal history, since no conflicting locks are granted on resource $r1$ at any point in the history. In addition, since T is not degenerate, it must GET or PUT resource $r1$ before the unlock at step j and must GET or PUT resource $r2$ after the lock at step k. From this $\langle T, \langle r1, j1\rangle, T'\rangle$ and $\langle T, \langle r2, j2\rangle, T'\rangle$ must be in the DEP(H). Hence $T <<< T' <<< T$, and T is a wormhole in the history H. Invoking the Wormhole Theorem, H is not isolated history. Intuitively, T sees resource $r1$ before it is updated by T' and sees resource $r2$ after it is been updated by T'; thus T is before and after T'. This is a concurrency anomaly.

7. Conclusions and future work

We have provided a RESTful framework for transactions by adapting the conventional locking mechanism to work within the architectural style of REST. We have shown that this locking mechanism is well-formed and sound. While this model can cover multi-service transactions by emulating 2PC, the full examination of such capabilities belongs in future work. Other extensions to this work include multiple owner transactions. Also the model can be extended to express transactions that include any HTTP operation rather than our current limited scope. Further plans include long-running transactions with relaxed ACID constraints.

8. Acknowledgements

This work was supported by the EU-FP6 funded project OPAALS Contract No 034824.

9. REFERENCES

[1] Astrahan, M.M. et al. A history and evaluation of System R. Communications of the ACM 24, 632-646, 1981.

[2] Bernstein, P.A., Hadzilacos, V., and Goodman, N. Concurrency control and recovery in database systems. Addison-Wesley, Boston, MA, USA, 1987

[3] Cabrera, L.F. et al. Web Services Atomic Transaction (WS-AtomicTransaction). Version 1.0, IBM developerWorks 2005, http://www-128.ibm.com /developerworks/library/specification/ws-tx/#

[4] Castro, P., and Nori, A. Astoria: A Programming Model for Data on the Web. Data Engineering, In IEEE ICDE 2008. pp. 1556-1559, 2008.

[5] Crockford, D. JSON: The fat-free alternative to XML. Proc. of XML 2006.

[6] Date, C.J. An Introduction to Database Systems. 5th Edition, Addison-Wesley, Reading, MA, USA, 1996.

[7] Fielding, R.T. Architectural Styles and the Design of Network-based Software Architectures. University of California - Irvine, 2000.

[8] R.Fielding, J.Gettys, J.Mogul, H.Frystyk, T. Berners-Lee., "Hypertext Transfer Protocol--HTTP/1.1. RFC 2616," The Internet Engineering Task Force, 1999.

[9] Gray, J. & Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1993

[10] Greenberg, S. & Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. ACM conference on Computer supported cooperative work, pp. 207-217, 1994

[11] Hadley, M. & Sandoz, P. JSR 311: Java api for RESTful web services. Technical report, Java Community Process, Sun Microsystems, 2007.

[12] Hoffman, P. & Bray, T. Atom Publishing Format and Protocol (atompub). IETF, 2006.

[13] Khare, R. & Taylor, R.N. Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. Proc. 26th Int'l Conference on Software Engineering (ICSE) 23, 428-437, 2004

[14] McGuffin, L.J. & Olson, G.M. ShrEdit: A Shared Electronic Work Space. University of Michigan, Cognitive Science and Machine Intelligence Laboratory, 1992

[15] Ramakrishnan, R. & Gehrke, J. Database Management Systems. McGraw-Hill Science/Engineering/Math, 2003

[16] Razavi, A., Moschoyiannis, S. & Krause, P. Concurrency Control and Recovery Management in Open e-Business Transactions. Proc. WoTUG Communicating Process Architectures (CPA 2007) 267–285, IOS Press, 2007.

[17] Richardson, L. & Ruby, S. RESTful Web Services. O'Reilly Media, Inc., 2007

[18] Sun, C., Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. Proc. Computer Cooperative Work. ACM 59-68, 1998

[19] Vinoski, S. WS-nonexistent standards. Internet Computing, IEEE 8, 94-96, 2004

[20] Vinoski, S. Demystifying RESTful Data Coupling. Internet Computing, IEEE 12, 87-90, 2008

[21] A. Razavi, S. Moschoyiannis, P. Krause. A Coordination Model for Distributed Transactions in Digital Ecosystems. In IEEE Digital Ecosystems and Technologies (IEEE-DEST'07), 2007

[22] P. Furnis and A. Green. Choreology Ltd. Contribution to the OASIS WS-TX Technical Committee relating to WS-Coordination, WSAtomicTransaction and WS-BusinessActivity. November 2005

[23] F.H. Vogt, S. Zambrovski, B. Grushko et al. Implementing Web Service Protocols in SOA: WS-Coordination and WSBusinessActivity. In Proc.7th IEEE E-Commerce Technology Workshops, pp. 21-26, 2005.

Authorized licensed use limited to: Shanghai Jiao Tong University. Downloaded on July 27,2010 at 11:41:21 UTC from IEEE Xplore. Restrictions apply.