RESTful Transactions Supported by the Isolation Theorems*

Amir Razavi, Alexandros Marinos, Sotiris Moschoyiannis, and Paul Krause

Department of Computing, FEPS, University of Surrey,
Guildford, Surrey, GU2 7XH, UK
{a.razavi,a.marinos,s.moschoyiannis,p.krause}@surrey.ac.uk

Abstract. With REST becoming the dominant architectural paradigm for web services in distributed systems, more and more use cases are applied to it, including use cases that require transactional guarantees. We propose a RESTful transaction model that satisfies both the constraints of transactions and those of the REST architectural style. We then apply the isolation theorems to prove the robustness of its properties on a formal level.

Keywords: REST, Transactions, Isolation Theorems, Locking.

1 Introduction

Representational State Transfer (REST) is a distributed computing architectural style first defined in 1999 by Roy Fielding [7] as an abstraction of the architectural style that had emerged in the World Wide Web. REST focuses on resources identified by names, a fixed number of methods with known semantics to manipulate those resources, hypermedia as a means of traversing the resources and statelessness in the interactions between client and server. REST has gained traction in addressing many common use cases for distributed systems [4],[13]. As is common with disruptive technologies, REST over HTTP is evolving to compete with WS-* in increasingly advanced scenarios. While REST has made great progress, the WS-* stack is currently the only standardized way to perform arbitrary transactions. A RESTful API has to resort to custom solutions of variable quality in order to address this issue. This paper aims to define a RESTful transaction model that is designed to operate over HTTP. We then apply the Isolation Theorem to prove the correctness of the model.

In terms of RESTful transactions, various approaches have been proposed. 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, when transferring funds between bank accounts, this approach proposes creating a 'transfer' resource to which new 'transfers' can be POSTed. While this approach can be very simple to implement at design time, it ties users to the ability of the developers to predict usage at design time. Furthermore, in scenarios where a large or even infinite variation of transactions and transaction types may take place, it is not reasonable to expect all the corresponding

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

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 394–409, 2009.

resources to have been designed beforehand. Other approaches [14] suggest extending REST to include mutex locks which would require extending HTTP as well. The alternative to these approaches [18] is to introduce locks on resources by modelling them as resources themselves. While this approach looks much more capable, the details of its implementation and its extension into transactions have neither been fleshed out nor proven. The general term 'Transaction' has been introduced by Gray [1] and is defined by the four properties contained in the ACID acronym. These properties guarantee that a system is maintained in a consistent state, even as transactions are executed within it concurrently. This includes 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, there is no need for concurrency. The transactions can simply be run in sequence. Real-world interactive systems however, often require execution of several concurrent transactions. Use cases such as distributed environments or dynamic allocation of resources to external developers illustrate this.

While transactions are concerned with the constraints of maintaining the 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; Hypermedia as the engine of application state. Our efforts are directed at creating a truly RESTful transaction model that satisfies both the constraints of REST while possessing the ACID properties. This paper is structured as follows. Section 2 examines classic transactional challenges that appear in distributed systems. Section 3 introduces Isolation theorems, which includes theorems that show the correctness of a transactional system by applying the necessary constraints. Section 4 applies the transaction model in a RESTful framework. In section 5 the proof of correctness of constraints is applied, by using classical isolation theorems.

2 Concurrency Challenges in RESTful HTTP

The classic view of isolation considers the transaction in terms of inputs and outputs [10],[6]. This means that transactions have *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 must examine its four fundamental 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. In this model, we approach POST purely as a creation operation and use it in the mechanics of the model to handle creation of

resources such as transactions, locks, and others. However, transactions that include POST operations are outside the scope of our model. 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. In the proposed model, PUT operations on pre-existing resources are the main operations that a transaction can execute, with only the Update aspect within scope. DELETE is used to remove the resource representation at the target URI. While they are used as part of the mechanics of the model, transactions which include DELETE operations are out of scope of the proposed model. From the above discussion it can be extracted that our model is concerned with transactions that are sequences of GET operations or PUT operations, specifically when used for updating resources. While other uses of the above verbs are of interest, this limited scope makes robust theoretical consistency proofs feasible. The limitation to GET and PUT applies only to the target resources, those that will remain after the transaction has committed. For interactions with the transactional resources, those that are created to enable the execution of the transaction, the full range of HTTP operations is utilised. As GET operations do not change the state of resources, when 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. The basic assumption is that a transaction knows what it is doing in terms of its internal data manipulation, meanwhile if it runs in isolation (without any concurrent transactions), it will manipulate its own resource state correctly. Therefore, sequential PUTs within the same transaction are not problematic [10], [2]. At the same time however, overlap between PUTs of one transaction and GET action 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 GETs operations as inputs of transactions and PUTs operations as output operations of them, this can be expressed as:

$$\boldsymbol{O_i} \cap \left(\boldsymbol{I_j} \cup \boldsymbol{O_j}\right) = \emptyset \ \textbf{for all} \ \boldsymbol{i \neq j} \tag{1}$$

By letting I_i be the set of resources accessed via GET by transaction T_i (its inputs), and O_i be the set of resources altered via PUT by transaction T_i (its outputs). Based on EQ.1, in the set of transactions$\{T_i\}$, when their outputs are disjoint from one another's inputs and outputs, they can run in parallel with no concurrency anomalies. Clearly by applying EQ.1 any transaction scheduler can work. Conventionally for applying EQ.1 each transaction should declare its Input-Output set, then a scheduler is able to compare the new transaction's need to all running transactions and in case of a conflict, initiation of the new transaction would be delayed until the conflicting transactions complete. This approach is called '*Static allocation*'. The computing complexity of analysing the inputs and outputs before running transactions causes a bottleneck on scalability. The approach has been abandoned in more modern transactional environments [2], [9]. The '*Dynamic allocation*' scheme has been introduced as the substituting approach. Under the prism of dynamic allocations, we can view transactions as sequence of operations on resources. A particular resource is subject to one operation at a time. Each operation of a transaction is either a GET or a PUT. Resources go through a sequence of versions as

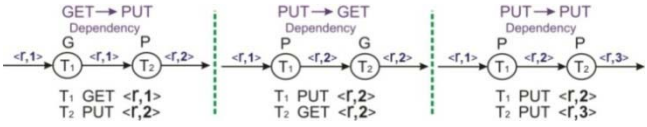

Fig. 1. Different Dependencies

they are updated by PUT operations. GETs do not change the resource version. If a transaction GETs a resource, the transaction depends on that resource version. If the transaction PUTs a resource, the resulting resource version depends on that transaction. When a transaction aborts and goes through the undo logic, all its PUT operations must be undone. These cause the resources to get new versions, as the undo looks like an ordinary new update. In the RESTful model we apply the shadow-based updating, which saves the complexity in terms of aborting the lock. This can be seen in the existence and behaviour of the conditional resource representation in section 4.

Theoretically a dependency graph can be read as a time sequence. The main conclusion of applying the ACID properties is that any dependency graph without cycles implies an isolated execution of transactions. General danger of violating isolation is related to the various dependency cycles. Similar to conventional transactions, REST cycle dependencies are categorised to three generic forms: 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). More details can be found at the classic references such as [10] or our previous work [21].

3 Isolation Theorems

Isolation theorems include several theorems, which shows the correctness of a transactional system by applying few constraints [10]. The constraints will be explained in sections and after applying them in a RESTful framework, we explain the proof of correctness of constraints, by using classical isolation theorems in section 5. In order to present a theoretical aspect of our model, we define a formal vocabulary that is larger than the standard HTTP operations. We call these formal terms operations. The correspondence with HTTP operations is made explicit in section 4.3 and figure x. More importantly, for avoiding violating consistent access, in term of GET and PUT resources, the SLOCK and XLOCK should be applied on the resources (before GET or PUT) and these locks should be released when the dependency on the resources expires. Therefore, the model should support the major *actions* of GET, PUT, XLOCK, SLOCK, UNLOCK on the resources, as well as generic actions BEGIN, COMMIT, ROLLBACK. GET and PUT have the usual meaning: GET returns the named resource's value to the program, while PUT alters the named resource's state. A *transaction* is any sequence of actions starting with a BEGIN action, ending with a COMMIT or ROLLBACK action, and containing any other BEGIN, COMMIT, or ROLLBACK actions. Figure 2 demonstrates an example in term of a conceptual transactional access to resources R1 and R2.

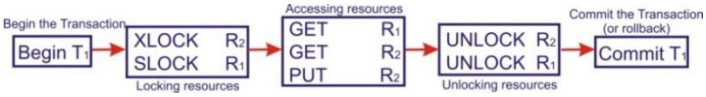

Fig. 2. Transaction life cycle

Transactions are characterized symbolically by a sequence such as$\langle\langle t, a_i, r_i\rangle | i = 1, \ldots, n\rangle$. This means that the i^{th} step of transaction t preformed action a_i on resource r_i.To simplify the transaction model, BEGIN, COMMIT, and ROLLBACK are defined in terms of other actions, so that only GET, PUT, LOCK, and UNLOCK actions remain. A *simple transaction* is composed of GET, PUT, XLOCK, SLOCK, and UNLOCK actions. Every transaction, *T*, can be translated into an *equivalent simple transaction* as follows [10]:

(1) Discard the BEGIN action.
(2) If the transaction ends with a COMMIT action, replace that action with the following sequence of UNLOCKS:

    <UNLOCK A | if SLOCK A or XLOCK A appears in *T* for any resource A>.

(3) If the transaction ends with a ROLLBACK statement, replace that action with the following sequence of PUTs and then UNLOCKs:

    <PUT A | if PUT A appears in *T* for any resource A> ||<UNLOCK A | if SLOCK A or XLOCK A appears in *T* for any resource A>.

The idea here is that the COMMIT action simply releases Locks, while the ROLLBACK action must first undo all changes to the resources the transaction wrote (PUT) and then issue the resources the transaction wrote (PUT) and then issue the unlock statements. If the transaction has no LOCK statements, then neither COMMIT nor ROLLBACK will issue any UNLOCK statements, as that would risk violating isolation. A transaction is said to be *well-formed* if all its GET, PUT, and UNLOCK actions are covered by locks, and if each lock action is eventually followed by a corresponding UNLOCK action [2], [9]. A transaction is defined as *two-phase* if all its LOCK actions precede all its UNLOCK actions. A two-phase transaction *T* has a growing phase, *T[1], …,T[j]*, during which it acquires locks, and a shrinking phase, *T[j+1], …,T[n]*, during which it releases locks [10]. The simplified Figure 3 (focusing on the formal locks), has been shown in Figure 3 and the concept of well-formed and two phase is indicated.

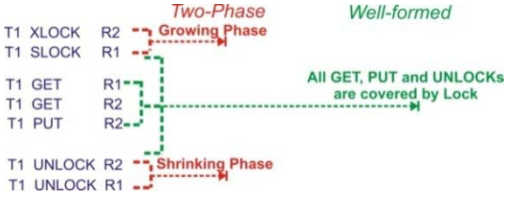

Fig. 3. Two-phase and Well-formed locking

Fig. 4. Different types of histories

First, a *history* is any sequence-preserving merge of the actions of a set of transactions into a single sequence for the set of transactions and is denoted $H = \langle \langle t, a, r \rangle_i | i = 1, \ldots, n \rangle$. Each step of the history $\langle t, a, r \rangle$ is an action a by transaction t on resource r. A history for the set of transactions $\{T_j\}$ is a sequence, each containing transaction T_j as a subsequence and containing nothing else. A history lists the order in which actions were successfully completed. *Serial histories* are One-transaction-at-a-time histories. In serial histories as no concurrency-induced, there is not any inconsistency and no problem with viewing dirty data by other transactions. As it is expected, a history should not complete a lock action on a resource when that resource is locked by another transaction. But if two or more transactions want to just read (GET) the content of a resource, they do not change the resource version (state). This may not cause any conflict or access to dirty data (data/resource which has been PUT by another transaction) but the transaction has not committed and may change the version of the resource again (2.2). The table 2 shows the lock compatibility. The locking compatibility rules constrain the set of allowed histories.

Legal history: Histories that obey the locking constraints are called *legal*. In Figure 4, three histories are shown, where History 1 and 2 are legal and History 3 is not. History 1 is a serial history. It is obviously legal, as each transaction will be run in sequence and no locks will conflict. History 2 is a non-serial legal history. There are no incompatible locks between T1 and T2 as T2 applies an XLOCK on resource B only when T1 has performed an UNLOCK. Finally, history 3 is a non-serial and not legal history, as resource B has an XLOCK by T1 but T2 applies an XLOCK on the same resource, which is illegal according to Table 1. As a consequence, we can see that T1 then performs a PUT based on its earlier GET and overwrites T2's PUT, which is the case of 'Lost Updates' as discussed in 2.

4 Locks in RESTful HTTP

Having defined the formal language we will use to prove the robustness of our model as well as discussed history well-formedness and legality, we now translate this abstract language into HTTP operations. To handle HTTP concurrency challenges, 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. For an API to be characterized as RESTful according to the hypermedia constraint, it must allow a client to interact

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

Fig. 5. (R) XML Fragment

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.

Ideally, any resource that can be served by an HTTP server should be a **Lockable Resource (R)** regardless of media type. This however would require the HTTP protocol to carry the metadata for the locking mechanism. Since we wish to preserve the HTTP protocol, we can use either prescribe that the resource links to lock collection and the transaction collection, or create custom HTTP headers that contain them. An example of the first approach can be seen in Figure 5. What is important is that the client has access to these resources while not obstructing normal use of the lockable resource.

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

Table 1. Elements of R-L

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.
ConditionalRepresentationURI: A link to the representation of the resource that will come into effect once the lock is committed.
InitialRepresentationURI: A link to the initial state of the lock resource.

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 the lock references. The time period 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 commit reacts as if a simple PUT or DELETE was applied. This was a specific design objective. Direct 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.

Table 2. Legal lock sequences

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

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 [13]. Since ATOM does not support sequencing entries, we use the 'PrevLockURI' element of the lock resource to create a linked list of locks that can be represented as an ATOM Feed. 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.

4.1 Two Phase Locking and Recoverability

Clarifying the scope of each transaction and determining whether it is in a GROWTH or SHRINK phase is necessary. In this part we introduce the required resources. The **Transaction (T)** resource is represented by a dedicated media type and should contain a TransactionCollectionURI, an OwnerURI and a TransactionLockCollectionURI. These 3 elements identify the collections of information vital to the execution of a transaction. The owner of the transaction can GET the transaction resource as a means of locating these collections. The **Transaction Collection (Tc)** 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. The **Transaction Lock Collection (T-Lc)** 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.

Table 3. Available Operations for T-Lc

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.

4.2 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 actual 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,

called a *conditional resource representation*. The **Initial Resource Representation (R-L-IR)** is of identical media type as the locked resource and stores the initial state. The initial representation is archived together with the lock to represent the change caused by the commit of the lock and enable rollback. The **Conditional Resource Representation (R-L-CR)** is of identical media type as the locked resource and is essentially the state that will be applied to the resource once the XLOCK is committed.

Table 4. Available Operations for R-L-CR

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.

4.3 Model Overview

Having defined all the resource types, it is easy to see that an interconnected network arises (Figure 6). 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 perforated 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. Table 5 summarizes the relevant resource types of our model.

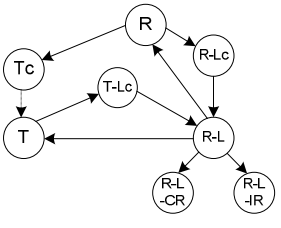

Fig. 6. Resource Hypermedia connections

Table 5. Transaction model resource types

Lockable Resource (R)	A resource that locks can be applied to.
Resource Lock Collection (R-Lc)	The collection of locks that apply to a particular resource.
Lock Resource (R-L)	The representation of a specific lock.
Conditional Resource Representation (R-L-CR)	The potential representation of a locked resource.
Conditional Resource Representation (R-L-IR)	The initial representation of a locked resource.
Transaction Collection (Tc)	The collection of transactions on the server.
Transaction Resource (T)	The representation of a specific transaction.
Transaction Lock Collection (T-Lc)	The collection of locks connected to a specific transaction.

4.4 RESTful Transaction Examples

To illustrate the operation of the transaction model, table 6 shows a scenario where two transactions from clients A and B interact with resources, R1 and R2. Table 7 shows what happens if a third client tries to XLOCK a resource that is already locked.

Table 6. Concurrent transactions

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 XLOCK 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 XLOCK 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
A	GET	R2-L1	200 OK	GETting R1 to extract location of R2-C
A	PUT <new representation>	R2-C	201 CREATED	ıting a conditional Representation of R2
A	DELETE	T1	200 OK	Commiting R2-C to R2 and Unlocking R1 and R2
B	GET	R2	200 OK	GETting R2 to extract location of R2-LC
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 representation>	R2-C	201 CREATED	Creating a conditional Representation of R2
B	PUT <new representation>	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

Table 7. Attempting to lock an already locked resource

Client	Operation	Resource	Response	Description
C	POST <LOCK {type:X}>	R2-LC	403 Forbidden	POSTing an XLOCK to R2-LC while R2 is locked

5 Applying the Isolation Theorems to REST

As our approach follows two-phase and well-formed locking, in this section, we use the classical isolation theorems [2], [9], [10] to show its correctness. Since the formal model introduced in section 3 is fully compatible with Isolation theorems, we apply it in classical proof of isolation theorems. For doing so, we can start from the main property of our model; all transactions are well-formed and two-phase. Based on the Locking Theorem, if all transactions are well-formed and two-phase, then any legal history will be isolated and based on Wormhole Theorem, A history is isolated if, and only if, it has no wormhole transactions. By adopting these two theorems, we show our approach, does not have any wormhole. We start by formulating the wormhole in the RESTful formal approach, then presenting the wormhole theorem and finally evoking the Locking theorem in our RESTful formal presentation (see Figure 7).

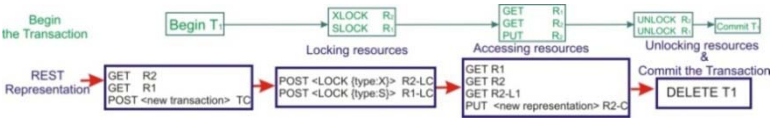

Fig. 7. Mapping Actions to Operations

5.1 Dependency and Wormholes

First we try to have a definition of a clear legal history; transaction t has resource r locked in SHARED mode at step k of history H, if for some $i < k$ action $H[i] = \langle t, SLOCK, r\rangle$, and if there is no $\langle t, UNLOCK, r\rangle$ action in the subhistory $H[i+1], \ldots, H[k-1]$,

similarly transaction t has resource r locked in EXCLUSIVE mode at step k is defined analogously. Then we say history h is legal if there is no step $H[k]$ of H at which two distinct transactions have the same resource locked in incompatible mode. In a simple way, we can say any data which has been PUT by the transaction is dirty data until is unlocked. Therefore when we analyse the system behaviour by using the history, easily we can say at each step of history, which resource value have been committed and which are dirty. We can analyse this by using dependency graph. One transaction instance T is said to depend 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'. We can formalise different dependencies (Fig 1) by Dependency Graph; a directed graph where nodes are 'transactions', Arcs are 'transactions dependencies' and label is 'resource versions'. The *version* of an resource r at step k of a history is an integer and is denoted $V(r,k)$. In the beginning each resource has version zero ($V(r,0)$=0). At step k of history H, resource r has a version equal to the number of writes of that resource before this step. Formally this means:

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

(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; Let $T1$ and $T2$ 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 $a1$ of $T1$ on resource r, step $H[j]$ involves $a2$ of $T2$ on r, and suppose there is no PUT of 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:

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

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 label 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' labelled by $\langle r, j\rangle$. Two histories are equivalent, if they have the same dependency relation.

The dependency of history defines a time order of the transactions. Conventionally this ordering is signified by $<<<_H$, (or simply by $<<<$), and it is the *transitive closure* of $<<<$. It is the smallest relation satisfying the equation:

$$T <<<_H T' \text{ if } \langle T, r, T'\rangle \in \text{DEP}[H] \text{ for some resource version } r, \text{ or}$$

($T <<<_H T''$ and $\langle T'', r, T'\rangle \in \text{DEP}[H]$ for some transactions T'', and some resource r). Off the record, $T <<< T'$ if there is a path in the dependency graph from transaction T to transaction T'. The $<<<$ ordering defines the set of all transactions that run before or after T;

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

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

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):

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

Serial histories do not have wormholes as all the actions of one transaction precede the actions of another; the first cannot depend on the outputs of the second.

Wormholes Theorem: Based on wormhole theorem, a history is isolated if, and only if, it has no wormhole transactions. On the other hand, the isolated histories have the *unique* property of having no wormholes. It proves a history that is not isolated has at least one wormhole; $T <<< T' <<< T$. In graphical term, if the dependency graph has a cycle in it, then the history is not equivalent to any serial history because some transaction is both before and after another transaction. (History 3 Fig 4). A wormhole in a particular history is a transaction pair in which T ran before T' ran before T. A history is said to be isolated if it is equivalent to a serial history. As the first part of the proof of the concept, the classical testimony of Wormhole theorem has been recalled [9],[10]; Isolated history has not any wormholes. This proof is by contradiction; Suppose H is an isolated history of the execution of the set of transactions $\{T_i | i = 1, \dots, n\}$. By definition, then, H is equivalent to some serial execution history, SH, for that same set of transactions. Without loss of generality, assume that the transactions are numbered so that $SH = T_1 \| T_2 \| \dots \| T_n$. Suppose, for the sake of contradiction, H has a wormhole; that is there some sequence of transactions $T, T', T'', \dots, T'''$ such that each is BEFORE the other (i.e., $T <<<_H T$), and the last is BEFORE the first (i.e., $T''' <<<_H T$). Let i be the minimum transaction index such that T_i is in this wormhole, and let T_j be its predecessor in the wormhole (i.e., $T_j <<<_H T_i$). By the minimality of i, T_j comes completely AFTER T_i in the execution history SH, so that $T_j <<<_{SH} T_i$ is impossible (recall that SH is a serial history). But since H and SH are equivalent, $<<<_H = <<<_{SH}$; therefore, $T_j <<<_{SH} T_i$ is also impossible. This contradiction proves that if H is isolated, it has no wormholes.

A history without wormhole is isolated: Our adopted proof (like the classic Wormhole theorem [10]) is by induction on the number of transactions, n, that appears in the history, H. The induction hypothesis is that any n transaction history H having no wormholes is isolated (equivalent to some serial history, SH, for that set of transactions). If $n < 2$, then any history is serial history, since only zero or one transaction appears in the history. In addition, any serial history is an isolated history. The basis of the induction, then, is trivially true. Suppose the induction hypothesis is true for $n - 1$ transactions, and consider some history H of n transactions that has no wormholes. Pick any transaction T, then pick any other transaction T', such that $T <<< T'$, and continue this construction as long as possible, building the sequence $S = \langle T, T', \dots \rangle$. Either S is infinite, or it is not. If S is infinite, then some transaction T'' must appear in it twice. This, in turn, implies that $T'' <<< T''$; thus, T'' is a wormhole of H. But since H has no wormholes, S cannot be infinite. The last transaction in S-call it T^*- has the property $\text{AFTER}(T^*) = \emptyset$, since the sequence

cannot be continued past T^*. Consider the history, $\mathrm{H}' = \langle\langle t_i, a_i, r_i\rangle \in H | t_i \neq T^*\rangle$. H' is the history H with all the formal actions (RESTful operations) of transaction T^* removed. By the choice of T^*,

$$\mathrm{DEP}(\boldsymbol{H}') = \{\langle \boldsymbol{T}, \langle \boldsymbol{r}, \boldsymbol{i}\rangle, \boldsymbol{T}'\rangle \in \mathrm{DEP}(\boldsymbol{H}) | \boldsymbol{T}' \neq \boldsymbol{T}^*\} \tag{2}$$

H' has no wormholes (since H has no wormholes, and $\mathrm{DEP}(H) \supseteq \mathrm{DEP}(H')$). The induction hypothesis, then, applies to H'. Hence, H' is isolated and has an equivalent serial history $SH' = T_1 \| T_2 \| \ldots \| T_{n-1}$ for some numbering of the other transactions.

The serial history $SH = SH' \| T_n = T_1 \| T_2 \| \ldots \| T_{n-1} \| T^*$ is equivalent to H. To prove this, it must be shown that $\mathrm{DEP}(SH) = \mathrm{DEP}(H)$. By construction,

$$\mathrm{DEP}(\boldsymbol{SH}) = \mathrm{DEP}(\boldsymbol{SH}' \| \boldsymbol{T_n}) = \mathrm{DEP}(\boldsymbol{SH}') \cup \{\langle \boldsymbol{T}', \langle \boldsymbol{r}, \boldsymbol{i}\rangle, \boldsymbol{T}^*\rangle \in \mathrm{DEP}(\boldsymbol{H})\} \tag{3}$$

By definition, $\mathrm{DEP}(SH') = \mathrm{DEP}(H')$. Using this to substitute equation EQ. 2 into equation EQ. 3 gives:

$$\mathrm{DEP}(\boldsymbol{SH}) = \{\langle \boldsymbol{T}, \langle \boldsymbol{r}, \boldsymbol{i}\rangle, \boldsymbol{T}'\rangle \in \mathrm{DEP}(\boldsymbol{H}) | \boldsymbol{T}' \neq \boldsymbol{T}^*\} \cup \{\langle \boldsymbol{T}', \langle \boldsymbol{r}, \boldsymbol{i}\rangle, \boldsymbol{T}^*\rangle \in \mathrm{DEP}(\boldsymbol{H})\} = \mathrm{DEP}(\boldsymbol{H}) \tag{4}$$

Thus, the identity $\mathrm{DEP}(SH) = \mathrm{DEP}(H)$ is established, and the induction step is proven. The wormhole theorem is the basic result from which all the others follow. It essentially says "cycles are bad". Wormhole is just another name for cycle. The wormhole theorem can be stated in many different ways. One typical statement is called the *Serializability Theorem*: A history H is isolated (also called a *serializable schedule* or a *consistent schedule*) if, and only if, $<<<_H$ implies a partial order of the transactions. (Alternatively: if and only if it defines an acyclic graph, or implies a partially ordered set [6]).

Locking Theorem: If all transactions are well-formed and two-phase, then any legal history will be isolated. As our RESTful framework, use the 'DELETE' operation (section 4), for unlocking resources (Shrinking phase), we can adopt the conventional proof [9] as bellow;

This proof is by contradiction. Suppose H is a legal history of the execution of the set of transactions, each of which is well-formed & 2-phase. For each transaction, T, define SHRINK(T) to be the index of the first unlock step of T in history H ; formally:

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

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 prove:

Lemma: if $T <<< T', then\ \mathrm{SHRINK}(T) < \mathrm{SHRINK}(T')$.

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 $\mathrm{DEP}(H)$). 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 schedule, and these locks would conflict, so there must be a *k*1 and *k*2, such that:

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

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. Having proved the lemma, the proof of the theorem goes as follows; Assume, for the sake of contradiction, that H is not isolated. Then, from the wormhole Theorem, there must be a sequence of transactions $\langle T_1, T_2, T_3, \dots, T_n\rangle$, 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 lemma, this in turn means that $\text{SHRINK}(T_1) < \text{SHRINK}(T_2) < \cdots < \text{SHRINK}(T_n) < \text{SHRINK}(T_1)$. But since $\text{SHRINK}(T_1) < \text{SHRINK}(T_1)$ is a contradiction, H cannot have any wormholes.

Locking Theorem (Converse): One may argue about the necessity of well-formed and two-phase history, which our approach warily follows. For proving the necessity of these properties, we use the converse locking theorem [9], [10]; if a transaction is not well-formed or not two-phase, then it is possible to write another transaction such that the resulting pair has a legal but not isolated history (unless the transaction is degenerate). When the classical proof [10], relies on actions on objects (read and write), we modelled the actions in term of RESTful classic operations (GET and PUT) and adopt the proof; first not well-formed history; Suppose that 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

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

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

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

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.

6 Conclusions and Further Work

By adopting conventional isolation theorem, we have provided a RESTful locking framework for business transactions which avoids inconsistency when dealing with a highly concurrent environment. While our detailed discussion shows the most important consistency issues are addressed, recoverability has been out of the scope of this paper. A recoverability extension on the RESTful transaction can be found in our paper at [22]. Meanwhile, long-running transactions and deadlock detection are the other issues which shall be considered as future work of this framework.

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., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing, Boston (1987)
3. Cabrera, L.F., et al.: Web Services Atomic Transaction (WS-AtomicTransaction). Version 1.0, IBM developerWorks (2005)
4. Castro, P., Nori, A.A.: A Programming Model for Data on the Web. Data Engineering, 2008. In: IEEE 24th International Conference on Data Engineering. ICDE 2008, pp. 1556–1559 (2008)
5. Crockford, D.: JSON: The fat-free alternative to XML. In: Proc. of XML 2006 (2006)
6. Date, C.J.: An Introduction to Database Systems, 5th edn. Addison-Wesley, Reading (1996)
7. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine (2000)

8. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: Hypertext Transfer Protocol–HTTP/1.1. RFC 2616, The Internet Engineering Task Force (1999)
9. Gray, J.: Benchmark Handbook: For Database and Transaction Processing Systems. Morgan Kaufmann Publishers Inc., San Francisco (1992)
10. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco (1993)
11. Greenberg, S., Marwood, D.: Real time groupware as a distributed system: concurrency control and its effect on the interface. In: Proceedings of the 1994 ACM conference on Computer supported cooperative work, pp. 207–217 (1994)
12. Hadley, M., Sandoz, P.: JSR 311: Java api for RESTful web services. Technical report, Java Community Process, Sun Microsystems (2007)
13. Hoffman, P., Bray, T.: Atom Publishing Format and Protocol (atompub). In: IETF (2006)
14. Khare, R., Taylor, R.N.: Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. In: Proc. of the 26th International Conf. on Software Engineering, vol. 23, pp. 428–437 (2004)
15. McGuffin, L.J., Olson, G.M.: ShrEdit: A Shared Electronic Work Space. University of Michigan, Cognitive Science and Machine Intelligence Laboratory (1992)
16. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill Science/Engineering/Math (2003)
17. Razavi, A., Moschoyiannis, S., Krause, P.: Concurrency Control and Recovery Management in Open e-Business Transactions. In: Proc. WoTUG (CPA 2007), pp. 267–285 (2007)
18. Richardson, L., Ruby, S.: RESTful Web Services. O'Reilly Media, Inc., Sebastopol (2007)
19. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proc. of the 1998 ACM conference on Computer supported cooperative work, pp. 59–68 (1998)
20. Vinoski, S.: WS-nonexistent standards. IEEE Internet Computing 8, 94–96 (2004)
21. Marinos, A., Razavi, A., Moschoyiannis, S., Krause, P.: RETRO: A Consistent and Recoverable RESTful Transaction Model. In: IEEE 7th International Conference on Web Services (ICWS 2009), Los Angeles, CA, USA (2009) (in the process to be published)