Journal of Web Engineering, Vol. 9, No. 3 (2010) 266–282

A TIMESTAMP-BASED TWO PHASE COMMIT PROTOCOL FOR WEB SERVICES USING REST ARCHITECTURAL STYLE

Luiz Alexandre Hiane da Silva Maciel
Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 Vila das Acácias
São José dos Campos, SP, CEP 12228-900, Brasil
luizhiane@gmail.com

Celso Massaki Hirata
Instituto Tecnológico de Aeronáutica
Praça Marechal Eduardo Gomes, 50 Vila das Acácias
São José dos Campos, SP, CEP 12228-900, Brasil
hirata@ita.br

Received June 21, 2009
Revised January 6, 2010

Service Oriented Architecture allows development of software with requirements of interoperability and weak coupling. Nowadays WS-* is the most used SOAP-based specification set for constructing web services. REST is an architectural style that permits the development of services in a simpler way than WS-* and obeys the SOA's paradigm, however, it does not provide standardized support to address some non-functional requirements of services, such as, security, reliability, and transaction control. This article proposes a REST-based technique to support the web services transactional control implementation. The technique uses the timestamp method and two phase commit protocol to control distributed systems transactions. An example of application using the technique is implemented to show its feasibility.

Keywords: Architectural style, concurrency control, REST, transaction, timestamp, web services, two phase commit protocol

1 Introduction

Service Oriented Architecture (SOA) has been gaining attention due to the ability to build interoperable and loose-coupled distributed applications. SOA enables software components reusability so that new applications can be developed by composing existing components. SOA promotes the interoperability between different platforms, systems and programming languages.

The two most adopted SOA implementations are: the WS-* and REST (Representational State Transfer). REST is both more recent and a "lighter" form to implement services than WS-*. WS-* is a set of specifications for the development of services based on SOAP [1] and WSDL [2]. These specifications were developed jointly by several organizations such as BEA, IBM and Microsoft. The specifications include *WS-Security* [3], *WS-Reliability* [4], *WS-Transaction* [5], *WS-Coordination* [6].

The WS-* specifications were designed to work together by using SOAP extensibility model. The specifications support the non-functional requirements implementation for applications that require a more complex infrastructure. In those applications aspects such as security and reliability can be essential to meet their business goals.

SOAP currently is more used to implement remote procedure call (RPC). In SOAP, the HTTP protocol is used to transport an XML document which holds information such as method invoked, input parameters and output expected [7]. SOAP envelope is transported inside HTTP envelope.

REST is an architectural style for web services implementation. Services implementation that follows REST makes extensive use of HTTP protocol characteristics. That form of implementation is called RESTful web services.

RESTful web services are simpler to understand and implement than those which adhere to the WS-*. However, many of the non-functional requirements in RESTful web services are not yet addressed, such as security, reliability and control of transactions. In the transaction control, in general, various resources have to be updated in a consistent way through distributed transactions. The goal is to ensure some form of ACID properties (atomicity, consistency, isolation, durability). However, not all of the properties can be ensured in web services context. For web services, there are alternatives to accomplish data consistency and integrity as the transaction compensation.

This work aims to provide a proposal to support transaction implementation in RESTful web services. The proposal uses timestamp, a non-lock concurrency control technique, and two phase commit protocol to implement web services using REST architectural style. We try to provide transactional support without adding complexity on the simplicity provided by the REST proposal [8].

Sections of this work are organized as follows: Section 2 proposes the use of timestamp transaction control to address the need to execute transactions involving RESTful services. Section 3 presents an analysis of the proposal and indicates some good practices raised for usage. Section 4 provides a comparative analysis between the proposed algorithm and the WS-Transaction. In Section 5, some conclusions and proposals for future work are presented.

2 Timestamp Concurrency Control for REST

This section presents a proposal to promote the transactional control for RESTful services through the use of non-lock concurrency control.

Subsection 2.1 provides an overview of REST. Subsection 2.2 introduces the timestamp concurrency control. In Subsection 2.3 it is proposed a way to use RESTful services with timestamp transactional control. In order to accomplish it, we extend the timestamp algorithm presented so that it can be used with web services. Subsection 2.4 shows a practical example using the proposed algorithm.

2.1 REST – Representational State Transfer

REST intends to evoke the image of how a well designed web application behaves: a network of web pages (a virtual state machine) where user interacts with the application by selecting links (states transitions), resulting in the transfer next page to him, which is rendered for his use [8].

The Web is composed of resources, which may be considered items of interest. For example, suppose that customers of a bank want to access information about their accounts. Thus, the bank can create a resource called *account/*{*id*}, where {*id*} is replaced by the client account number. Therefore, the URL for a customer with the account number 12345 is: `http://www.bank.com/account/12345`

Upon requesting that URL, a resource *representation* is returned, for example, *account-12345.html*. The client application is placed in a *state*, which displays the information to the user through a web browser. The user can click on hyperlinks within account12345.html and access other resources available on the bank's server, such as a page to make online payments or a resource that represents the savings account.

Other representations are accessed, taking the client application into new states. User can continue interacting indefinitely. Therefore, the client application changes (*transfers*) its state for each resource representation received. Hence, it emerges the definition of Representational State Transfer - REST.

According to Fielding [8], the REST key abstract information is a *resource*. Any information that may be appointed can be a resource: document, image, service, collection of other resources, non-virtual object and so on.

In SOA context, resource is a conceptual entity that identifies a service exposed to clients. It is usually a noun, since the verbs better indicate the action on the resource and not their identification.

2.1.1 Architectural Style

REST is an architectural style, it is not a standard. It does not make much sense to create a specification for REST because it is just a style that has to be understood to design web services in that style [9]. It is possible to make an analogy with the client-server architecture style, widely known. There is not a specification called client-server, but only rules that must be followed by applications that wish to follow that style.

An architectural style is a coordinated set of architectural constraints that restrict the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

According to Fielding [8], REST has a series of architectural restrictions that, when applied as a whole, emphasize scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

Some of main features proposed by REST are: (i) Client-Server paradigm: the important principle is the *separation of concerns* between the client and server, allowing them to evolve independently. (ii) Stateless: communication between client and server must be stateless. The server does not store conversational state with clients. Session state is kept entirely on the client. (iii) Cache: the responses to prior requests can be reused in response to later requests that are equivalent and likely to result in a response identical to that in the cache if the request were to be forwarded to the server. (iv) Uniform Interface: a central feature of REST is the uniform interface between the components. The overall system architecture is simplified and there is an improvement in visibility of interactions. (v) Layered System: system hierarchically organized in which each layer provides services for the upper layer and

uses those of the lower layer. (vi) Code on demand: the client can extend its functionality by downloading and executing code from the server as the client needs. Thus, the client is simplified since it reduces the number of features required to be pre-implemented.

While REST is not a standard, RESTful services implementation uses standards, such as HTTP, URL, XML, HTML, GIF, JPEG. In general, Web can be considered as a REST system [9]. Many of the services used in the Web, such as search services, booking-ordering services, dictionary services can be considered RESTful web services, at least partially.

Richardson and Ruby [7] define a new term to indicate a concrete architecture that implements the REST architectural style called the *Resource-Oriented Architecture - ROA*. Basically, it makes use of URI, HTTP and XML. ROA is an implementation of REST, which in turn complements SOA. Therefore, ROA is only a term coined by Richardson and Ruby to make clear the difference between REST and a concrete implementation of that architectural style.

According to them [7], in order to identify a resource, ROA uses *URI - Universal Resource Identifiers* [10]. URI is the name and the address of a resource. If a piece of information does not have a URI, it is not a resource and may not be directly accessed on the web, because the URI is the entity that allows the access of such a piece of information providing a single address. It is important that URIs have a clear correspondence with the resources they identify.

ROA proposes the use of HTTP fundamental methods for the most common operations to address the need for a uniform interface [7]. The common operations are referenced as CRUD (create, retrieve, update, delete). The fundamental HTTP methods include: GET, PUT, POST, and DELETE. GET retrieves information about a resource. PUT creates a new resource when new URI is provided by the client, i.e., the client defines the URI that will become available. PUT also updates an existing resource. POST creates a new resource without providing new URI, i.e., the service itself defines the URI that will become available. DELETE erases an existing resource.

2.1.2 Transactions as Resources

In ROA each HTTP request has one resource as a destination, but some services exposes operations that generate multiple resources, such as the operation of transferring funds from a checking account to a savings account.

In order to solve this problem, Richardson and Ruby [7] propose to expose the transactions as resources. Suppose that the checking account resource is exposed in `/checking/11` and the savings account resource is exposed in `/savings/55`. Both accounts have balance of $200 and someone wishes to transfer $50 from the checking account to the savings account. The following steps are proposed:

1 Client creates a transaction by sending a POST to a transaction factory resource. A possible correspondent HTTP message is:

```
POST /transactions/account-transfer HTTP/1.1
Host:  bank.com
```

2 The response provides the URI for the created resource. A possible correspondent HTTP message is:

```
201 Created
Location:  /transactions/account-transfer/11a5
```

3 Client submits the first part of the transaction which updates the balance of the checking account by sending a HTTP message like:

```
PUT /transacations/account-transfer/11a5/checking/11 HTTP/1.1
Host:  bank.com
balance=150
```

4 Client submits second part of the transaction which updates the balance of savings account by sending a HTTP message like:

```
PUT /transacations/account-transfer/11a5/savings/55 HTTP/1.1
Host:  bank.com
balance=250
```

5 At any time it is possible to make a rollback by sending a DELETE to the transaction URI.

6 Client sends a request to commit the transaction. A possible correspondent HTTP request message is:

```
PUT /transacations/account-transfer/11a5 HTTP/1.1
Host:  bank.com
committed=true
```

7 The server must ensure that the transaction maintains the resource in a consistent state. If everything is successfully executed, the transaction is committed and the resources updated. A possible response from the server is:

```
200 OK
Content-Type:  application/xhtml+xml
<a href="/checking/11"> Checking Account #11</a>:  New balance $150
<a href="/savings/55"> Savings Account #55</a>:  New balance $250
```

In the server, the procedure is implemented by a component that receives the various requests and creates a queue of actions associated with the transaction. When the server is requested to commit, it initiates a transaction in the database, applies the actions in the queue, and then commits the transaction. In case of failure, it is propagated as a transaction commit failure [7].

2.2 Timestamp-Based Concurrency Control

In timestamp-based transaction, a unique timestamp is assigned to each transaction. Transactions are processed so that their execution is equivalent to a serial execution in timestamp order [11]. Timestamp ordering is a technique whereby a serialization order is selected a priori and the transaction execution is forced to obey this order [12].

The timestamp defines the transaction position in the time sequence of transactions. The timestamp ordering rule is based on operation conflicts and is simple: *A transaction's request to write an object is valid only if that object was last read and written by earlier transactions.*

A transaction's request to read an object is valid only if that object was last written by an earlier transaction [13].

The timestamp concurrency control does not use locks to preserve the integrity and consistency of the application. Each timestamp value is unique and accurately represents an instant in time. No two timestamps can be the same. A higher-valued timestamp occurs later in time than a lower-valued timestamp.

Whenever a transaction starts, it receives a timestamp so that it is possible to predict the transactions executions order. If two distinct transactions affect the same object, the transaction that has the earlier timestamp must be applied before the other one. However, if the wrong transaction is actually presented first, it is aborted and must be restarted, probably with a higher timestamp.

2.2.1 Basic Timestamp Mechanism – BTS

This subsection presents the timestamp mechanism for concurrency control of distributed database transactions. The timestamp mechanism replaces the lock mechanism and exploits the optimistic execution of the transactions.

Every data object has a read timestamp, which is updated whenever the object's data is read, and a write timestamp, which is updated whenever the object's data is changed.

The basic timestamp mechanism applies the following rules, accordingly to Ceri and Pelagatti [11]:

```
1 - Each transaction receives a timestamp when it is initiated at its site
    of origin;
2 - Each read R or write W operation which is required by a transaction has
    the timestamp of the transaction;
3 - Each data item (x) contains the following information:
     (i) WTM (x) - the largest timestamp of a write operation on x;
    (ii) RTM (x) - the largest timestamp of a read operation on x;
4 - Let TS be the timestamp of a read operation R on data item x;
        If TS < WTM(x) then
           reject R and restart the transaction with a new timestamp;
        else
           execute R and RTM(x) = max(RTM(x), TS);
5 - Let TS be the timestamp of a write operation W on data item x;
        If TS < RTM(x) or TS < WTM (x) then
           reject W and restart the transaction with a new timestamp;
        else
           execute W and WTM (x) = TS;
```

Rules 4 and 5 ensure that conflicting operations are executed in timestamp order at all sites; hence the timestamp order is a total order satisfying the *serializability* [11, 14] condition, and the executions produced by this mechanism are correct.

The basic timestamp mechanism is deadlock-free, because transactions are never blocked [11]. If a transaction cannot execute an operation, it is restarted. However, we have to cope with the cost of restarting transactions.

2.2.2 Timestamp with Two Phase Commit Protocol - TS2PC

This subsection describes the timestamp mechanism for the two phase commit protocol.

The timestamp algorithm described is sufficient to ensure the serializability of transactions, however, it needs to be integrated with 2-Phase-Commit protocol (2PC) to ensure atomicity [11]. 2PC requires a time interval during which participating transaction agents can abort or commit. Using a locking mechanism this is achieved by holding all exclusive locks until the transaction ends (committing or aborting). No transaction can read the data which has been written by a not yet committed transaction. With the timestamp mechanism, a different solution is required: instead of exclusive locks, *prewrites* are used [11].

Thus, 2PC can be incorporated in BTS algorithm by using *prewrites* and accepting or rejecting prewrites instead of writes. They are buffered and not applied directly to the data items. Only when the transaction commits, the corresponding write operations are applied to the data. Prewrites cannot be converted in write operations if the transaction is aborted.

Once a prewrite has been accepted, it is guaranteed to accept the corresponding write no matter when the write arrives. Write operations will not be rejected. Once a $prewrite(x)$ with timestamp TS is accepted, any $read(x)$ (or $write(x)$) with timestamp greater than TS cannot be allowed until the $write(x)$ is executed [14].

The timestamp algorithm described above must be slightly modified to account for the use of prewrites. To do so, rules 4 and 5 are substituted by the following rules 4, 5, and 6.

4 - Let TS be the timestamp of a prewrite operation PW_i on data item x;

```
If TS < RTM(x) or TS < WTM (x) then
   reject  PWi and restart the transaction;
else
   put the  PWi  and its TS into the buffer;
```

5 - Let TS be the timestamp of a read operation R_i on data item x, and $TS(PW-min)$ the lower timestamp of any prewrite in the buffer;

```
If TS < WTM(x) then
   reject Ri and restart the transaction;
else // TS >= WTM (x)
     If (no PWi in the buffer)
        execute Ri and RTM(x)= max(RTM(x), TS);
     else
          If TS <= TS(PW-min) then
             execute Ri and RTM(x) = max(RTM(x), TS);
          else // there is one (or more) PW with TS(PW) < TS
             Ri is buffered until all transactions which
             has TS(PW) < TS commit;
```

6 - Let TS be the timestamp of write operation W_i on data item x. This operation is never rejected; however, it is possibly buffered if there is a prewrite operation $PW(x)$ with $TS(PW) < TS$. W_i will be executed and eliminated from the buffer when all prewrites with smaller timestamp have been eliminated from the buffer.

The reason R_i is buffered, in rule 5, is that the write operation $W(x)$ corresponding to the prewrite $PW(x)$ cannot be rejected. Therefore, we must avoid $TS(W) < RTM(x)$. But $TS(W) = TS(PW)$, because they are issued by the same transaction; we must avoid applying R_i since the value of $RTM(x)$ would be set equal to the value of TS, thus making $W(x)$ impossible. The R_i is executed and eliminated from the buffer when no more PW with smaller timestamp than R_i are pending on x. The reason for buffering W_i in rule 6 is similar to which has just been described for read operations.

It is worth noting that the use of prewrites is equivalent to applying exclusive locks on data items for the time interval between the prewrite and the commitment (write) or abort of the issuing transaction [11].

2.3 Timestamp-based Two Phase Commit Protocol for RESTful Services

In this section, we describe a procedure to address concurrency control with RESTful services. For this purpose, we use a timestamp based method to control concurrent access to REST resources.

The procedure exposed in Subsection 2.1.2 assumes that all resources participating in a transaction are present in the same server. On the server, the requests (HTTP PUT) are stored in the actions queue. When a client sends a commit request (HTTP PUT with *committed = true*), the server delegates transaction execution to a database. The server starts a database transaction, applies all queued actions, and commits the database transaction.

If the transaction is distributed, i.e., if the operations are executed between different banks, considering the transferring funds example, with different URIs and databases systems, the proposal does not work because it assumes that there is a single database responsible for executing the transaction. So, it does not ensure the ACID properties with the distributed operations. For example, assuming that there are two servers, each one with its own database, and the server that holds the checking account is not operational, the transaction executes only the credit operation on the savings account, which contradicts the atomicity property.

Therefore, the procedure of transactions in Subsection 2.1.2 does not support distributed transactions control. The services operate as a facade for the actual transaction operations implementation; one can say the services are just interfaces to a database transaction.

It is possible to propose new ways to deal with distributed transactions in RESTful services. The proposal presented in this section uses a timestamp based method. Thus, from now on we describe the approach for dealing with concurrency control in the RESTful services domain.

2.3.1 Timestamp-based 2PC Protocol for Web Services using REST Architectural Style – TS2PC4RS

The timestamp algorithm described in Section 2.2.2 (TS2PC) is correct. However, it is not practical in a service domain as it can cause long waiting time for read operations. When a client sends a read operations to a REST resource, the ideal is it receives an immediate response and does not have to wait until the read can be executed. For write operations it is not so critical because when a client sends a write operation W to a REST resource, the server accepts W, returns a success message to the client, and processes W when all prewrites with smaller timestamp pending on REST resource commits or aborts.

In the web services domain a flexible approach should allow more concurrent accesses to the services. The concurrent execution control of transactions can be achieved through the application domain characteristics. Web services transactions can have long duration and can involve different organizations, each one with its own business rules.

We propose a new algorithm based on the TS2PC described in Section 2.2.2. Our algorithm (TS2PC4RS) uses timestamp technique with the 2PC protocol.

The 2PC protocol has a *coordinator* site and the *agents* sites. In a transaction, the coordinator site sends requests to agents sites. Requests are sent through prewrites. If all agents sites can perform the operations, they reply with the message *ready* to the coordinator site. If the coordinator site receives ready from all the agents, the transaction is committed. So, the coordinator sends the *commit* message (decision to commit) to all agents.

If an agent site cannot perform the operation requested, it sends *not-ready* message to the coordinator. If coordinator receives a not-ready message, the transaction must be aborted (decision to abort). So, the coordinator sends *abort* message to all agents.

Therefore, agents determine whether the prewrites are accepted based on the application business rules. Several prewrites may exist for the same data item. The prewrites may or may not be completed. It is a coordinator decision. The agent, through its support application, ensures local consistency of its resources (data items) and the prewrites conversion into writes in case the coordinator decides to commit the transaction. Agent also ensures prewrite discard if the coordinator does not commit the transaction by sending abort messages to agents.

Next, we describe the TS2PC4RS algorithm for concurrency control in RESTful services domain. One of the major differences is that, in certain cases, the execution of a read operation R may return

not only the data item value updated at WTM, but also a sub-list of data items in the prewrite buffer and its timestamps.

```
Every write operation W is preceded by prewrite PW.

  1 - A unique timestamp is assigned to each transaction in their origin;
  2 - Each read R, write W, and prewrite PW operation has the transaction's
      timestamp TS;
  3 - Each data item (x) contains the following information:
     (i) WTM (x) - the largest timestamp of a write operation  on x;
    (ii) RTM (x) - the largest timestamp of a read operation  on x;
   (iii) LPW(x) - a list of buffered prewrites on x in timestamp order;
  4 - For prewrite operations:
        If TS < RTM(x) or TS < WTM (x) or PW places the data item
        in a inconsistent state then
           reject the PW operation and restart the transaction;
        else
           put the PW operation and its TS into the LPW;
  5 - For read operations R with timestamp TS:
        If TS < WTM(x) then
           reject R and restart the transaction;
        else // TS >= WTM (x)
             If (LPW is empty)
                execute read and RTM(x)= max(RTM(x), TS);
             else
                  If TS < TS(first(LPW)) then
                     execute read and RTM(x)= max(RTM(x), TS);
                  else
                     execute read and return the data item value committed at WTM,
                     WTM, and LPW sub-list until TS;
```

If LPW is not empty, and an agent receives a read operation R with $TS(R) > WTM$ and $TS(R) >= TS(first(LPW))$, we cannot execute the read operation because there are transactions in progress (not committed yet). The value of the data item being read at $TS(R)$ cannot be accurately determined. However, considering a web service domain, it is worth to return an updated view of the data item. So, in this situation, we make the TS2PC4RS more flexible by returning the updated view of the data item, which contains the data item value in last committed write, the WTM, and a LPW sub-list which contains all PW with timestamp less than or equal to $TS(R)$. We can also return some computed value based on the LPW sub-list. For example, considering that all pending prewrites commit, it may be interesting to return the data item value with the updates, which will have $WTM = last(LPW_{sub-list})$.

Hence, the client, based on the *data item updated view* and its business rules, can decide what he wants to do. For example, depending on what the client is performing, it can abort the transaction, can try to predict the data item value at de $TS(R)$ (if it is not already calculated by the agent), or can wait sometime and re-send R.

When the transaction is committed, the operation W is performed in the data item, its corresponding PW is removed from the LPW, and $WTM = TS(PW_{removed})$. If the transaction is aborted, the PW is just removed from the LPW. So when the coordinator sends write operations W to all the participating agents, W must have an associate PW (with the same TS) previously sent to the agents. The following procedure must be accomplished when the agents receive the write operation W indicating the transaction commitment.

```
Search by the PW of the committed transaction in LPW;
If it is the first in LPW then
   Execute W, remove PW from LPW and WTM(x)=TS(removed PW);
   If there is a sequence of PWs marked for update-data in the LPW,
   immediately after the removed PW then
      Remove that sequence, execute the respective writes, and
      WTM(x)=TS(last PW of the removed sequence);
else // it is second forth
   Mark the PW for update-data;
```

A sequence of PW marked for update-data means that there is a LPW sub-list that can be removed from LPW if the corresponding updates are made in the data item.

If the transaction is aborted, the following procedure must be performed by the agents.

```
Remove PW from LPW;
If the removed PW was the first in LPW then
   If there is a sequence of PWs marked for update-data in the LPW,
   immediately after the removed PW then
      Remove that sequence, execute the respective writes, and
      WTM(x)=TS(last PW of the removed sequence);
```

An important note about the 2PC protocol is its resilience to certain failures [11], such as loss of messages and host failures, but to do so, the 2PC *timeouts* must be consistently set.

In the next Subsection, we describe an example using our proposal. The example represents the practical use of the TS2PC4RS in the realm of web services, since it tries to compose two different services provided by different players in order to achieve a new kind of service. Clearly, the services are implemented to work separately, but it is possible to use them in a joint manner.

2.4 Purchase of Tickets Example

In order to demonstrate the feasibility of the proposed timestamp transactional control, an example was implemented. The example considers two operations: purchase of tickets for a basketball game and purchase of train tickets to go to the game. Two clients try to execute transactions concurrently.

RESTful services receive requests from clients, which control the transaction executions. Services can process several requests, even though there are some transactions in progress. The client's objective is to buy a certain number of tickets for the game together with the train tickets to go to the game. So, clients have to buy the same amount of tickets for the game and for the train seats.

Each REST resource that uses the timestamp concurrency control has the following additional attributes within its representation: its largest write operation timestamp (WTM), its largest read operation timestamp (RTM), and its buffered prewrites (LPW). The values of the attributes are defined according to the algorithm described in Section 2.3.1 (TS2PC4RS). When a resource is created, an initial value is assumed to each of these attributes.

We must make a mapping between the timestamp, 2PC, and REST concepts in order to use the proposed timestamp algorithm. The RESTful services (REST resources) implement the *2PC Agents*, controlling the access to the data items. The REST clients implement the *2PC Coordinator*, so they are responsible to start, control, commit, and abort transactions.

The client sends read operations (R) through HTTP GET messages; prewrite (PW) and write (W) operations through HTTP PUT messages. If the service executes R, a resource representation is returned with the HTTP 200 status code (OK). Otherwise if the service cannot process R, it returns a message with the HTTP 400 status code (Bad Request).

If the service successfully executes PW or W, a message with the HTTP 200 status code is returned. Otherwise if the service cannot process PW or W, it returns a message with the HTTP 400 status code.

We expose two resource types, namely the resource representing the number of available tickets and the resource representing the clients tickets booking, i.e., the purchase not yet committed. For

example, the REST service responsible for the game tickets sell is exposed at URI `/ticketsforgame/{timestamp}`; and its corresponding reservation is exposed at URI `/ticketsforgame/booking/{timestamp}`. The `{timestamp}` represents the timestamp at which the client wants to read or update the corresponding resource. We decide to expose, in addition to the data item (number of tickets), its LPW (reservation) in its own URI in order to allow clients to request reservations and track them. Table 1 provides an overview of the main resources, URIs, and operations for the service responsible for the game tickets. The service responsible for the train tickets have similar resources, URIs, and operations.

Table 1. The main resources, URIs, and operations for the Purchasing of Tickets Example.

Resource	URI	Method	Description
Tickets for game	`/ticketsforgame/{TS}`	GET	Retrieve the available tickets number
	`/ticketsforgame/{TS}`	PUT	Commit or abort the prewrite (reservation) at TS
Tickets Booking	`/ticketsforgame/booking/{TS}`	PUT	create a booking at TS
	`/ticketsforgame/booking/{TS}`	GET	Retrieve the status of the booking created at TS

Figures 1, 2, 3 illustrate the purchase of tickets scenario described above. The filled arrows represent the requests made by clients. The dashed arrows represent the responses sent by the servers to clients. The requests are numbered to indicate the order of execution and the responses indicate whether there is success or failure in processing the requests. The goal of client 1 is to buy 400 tickets in both services; and the goal of the client 2 is to buy 300 tickets in both services.

Server A hosts the RESTful service responsible for the game tickets, and the initial values for its attributes are: $tickets = 1000$, $WTM = (10, x)$, $RTM = (20, x)$, and $LPW = [\,]$. Server B hosts the RESTful service responsible for the train tickets, and the initial values for its attributes are: $tickets = 500$, $WTM = (15, x)$, $RTM = (30, x)$, and $LPW = [\,]$.

The timestamp is a record composed of two values: a positive integer that represents the timestamp – TS, and a coordinator identifier – CID, which is used to break ties when two transactions have the same timestamp. Thus each coordinator must have a unique ID, and a total order among all CID is assured. The client 1 initial timestamp is $(32, a)$, and the client 2 initial timestamp is $(40, b)$.

The first step of each client is to obtain the representation of resources involved by sending an HTTP GET request to the servers. The request URI to server A is `ticketsforgame/{timestamp}` and to server B is `ticketsfortrain/{timestamp}`. Both clients obtain the representations of available game and train tickets. See Figure 1-i.

A possible HTTP request message to retrieve the available tickets for the game can be:

```
GET /ticketsforgame/40b HTTP/1.1
Host: serverA.com
```

Below it is illustrated a response to the above request.

```
HTTP/1.1 200 OK
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ticketsforgame>
     <ticketsavailable>1000</ticketsavailable>
</ticketsforgame>
```

Both clients, who have timestamps greater than the corresponding WTMs, receive the available tickets for the game and for the train seats. LPW is empty, thus read operations are executed and the RTM is updated with the value of the expression $max(RTM, TS)$.

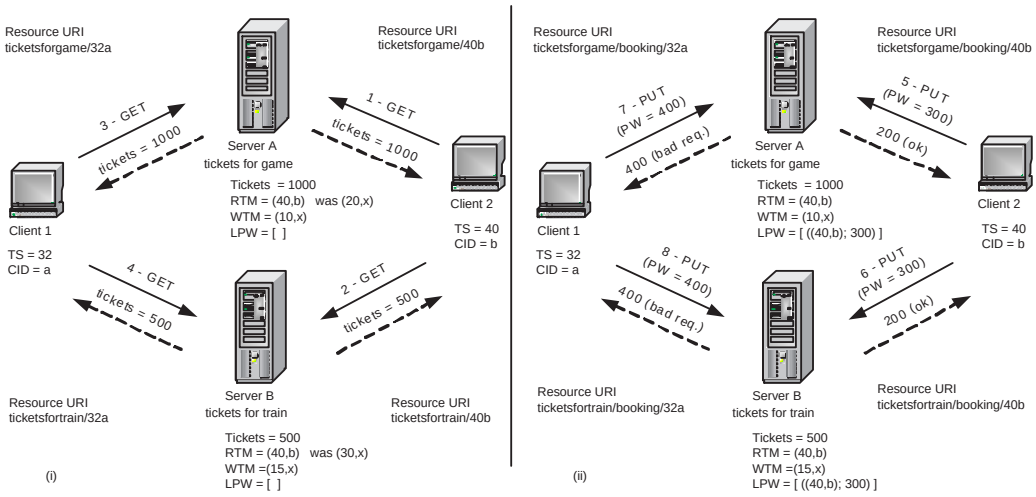

Fig. 1. (i) Each client retrieves the accounts representations. (ii) Only client 2 succeeds in sending prewrites.

With the representations, clients see if it is possible to book the desired number of tickets and send the PW messages, using the HTTP PUT, to the corresponding URIs. As illustrated by Figure 1-ii, in order to send a prewrite to server A, the URI `ticketsforgame/booking/{timestamp}` is used, and to server B, the URI `ticketsfortrain/booking/{timestamp}` is used.

At the time of sending the prewrites to the servers, client 1 cannot achieve any successful prewrite, because the RTM of both resources are greater than client 1's timestamp. Client 1, therefore, receive two *not-ready* messages in return to its prewrites and client 1 must restart its transaction with a higher timestamp. See Figure 1-ii. On the other hand, client 2 successfully books its tickets and receives *ready* messages from the servers. So, the server A has its LPW updated by the insertion of $((40, b); 300)$. The RTM of the data item for game tickets contains $(40, b)$ and the WTM remains untouched. The server B has its LPW updated by the insertion of $((40, b); 300)$. The RTM of the data item for train tickets is $(40, b)$ and the WTM remains untouched.

A request content to book 300 tickets for the game can be as follow.

```
PUT /ticketsforgame/booking/40b HTTP/1.1
Content-Type: application/xml
Host: serverA.com
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ticketsforgame>
     <booking>300</booking>
</ticketsforgame>
```

Because client 1 receives the *not-ready* messages from both servers, it restarts its transaction with a higher timestamp, $TS = (50, a)$, and sends HTTP GET requests to retrieve the resources representations again. But now, as it is shown in Figure 2-i, the LPWs are not empty, they contain one prewrite with $TS(PW) = (40, b)$. So, both servers cannot execute the actual read and return the data item updated view which contains the data item value at WTM, the WTM, and the LPW sub-list with the single element in LPW, which has timestamp less than $(50, a)$.

The feature of allowing the return of a resource updated view, including the transactions in progress, makes our proposal flexible, because it transfers to the client the responsibility to decide what the client wants to do, considering that some transactions are not committed yet. The client

decision, in most cases, can take into account the application domain where the transaction is inserted, i.e., the business rules that guide the interactions between clients and RESTful services.

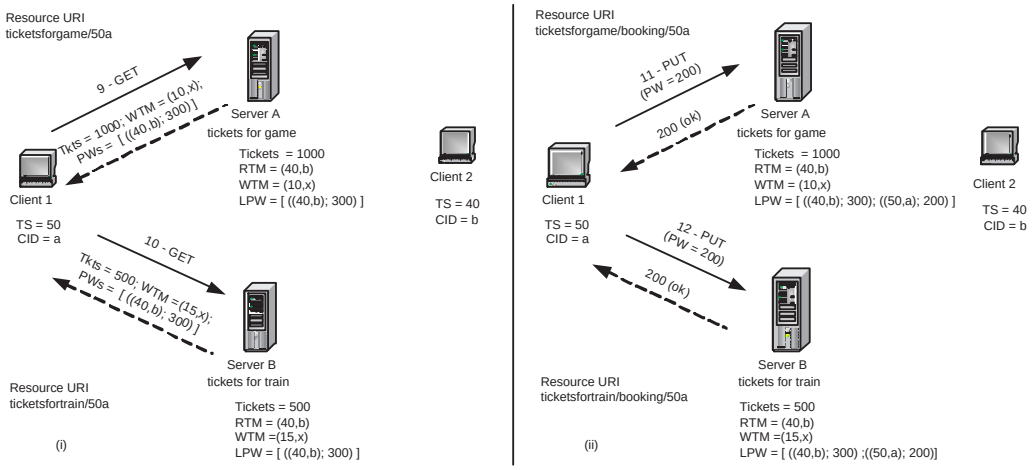

Fig. 2. (i) Client 1 restarts and requests the resources representations again, but there are transactions in progress. So some more information are sent back to client 1. (ii) Client 1 now succeeds in sending prewrites.

So, client 1, with the resources updated views, can decide based on its business rules. Let us assume client 1 verifies that there are transactions in progress which, if they are committed, they will not allow the purchase of the 400 tickets. So, client 1 decides to change its request, decreasing the amount of tickets from 400 to 200. If anyway, client 1 sends the request to buy 400 tickets, the server 2 rejects the prewrite since the purchase may leave the amount of tickets in an inconsistent state.

Thus, client 1 successfully books its 200 tickets and receives ready messages from the servers. As illustrated in Figure 2-ii, both servers have their LPW updated by the insertion of $((50, a); 200)$.

At this point, either client can commit its transaction. Assuming that client 1 sends the commit message first. The resources receive the write messages from client 1 and must accomplish the procedure described in Section 2.3.1. As, in the servers A and B, the PWs to be committed are the second in the LPWs, they are just marked for update-data in each resource. In Figure 3-i the asterisk represents the update-data mark.

When client 2 commits, according to the commitment procedure (Section 2.3.1), the resources execute the write operation, remove the committed PW from LPW, and update WTM with $TS(PW_{removed})$. The resources also remove the sequence of PWs marked for update-data, execute the respective writes, and update WTM with TS(last PW of the removed sequence). The values of all attributes updated in this case are shown in Figure 3-ii.

Another option of both clients is to abort the transaction. In this case, instead of sending a commit message to the resources, a client sends an abort message. The resource that receives the abort message must remove the corresponding PW from the LPW and, if there is a sequence of PWs marked for update-data in the PW, the sequence is removed in the same way described earlier.

3 Analysis of the Proposed Timestamp Technique

Through the proposed algorithms, some of the ACID properties can be assured when two transactions request two REST resources. Atomicity is achieved due to the use of 2PC jointly with timestamp concurrency control. So, or every transaction operations perform successfully or the transaction is aborted.

Consistency ensures data remains in a consistent state before the start of the transaction and

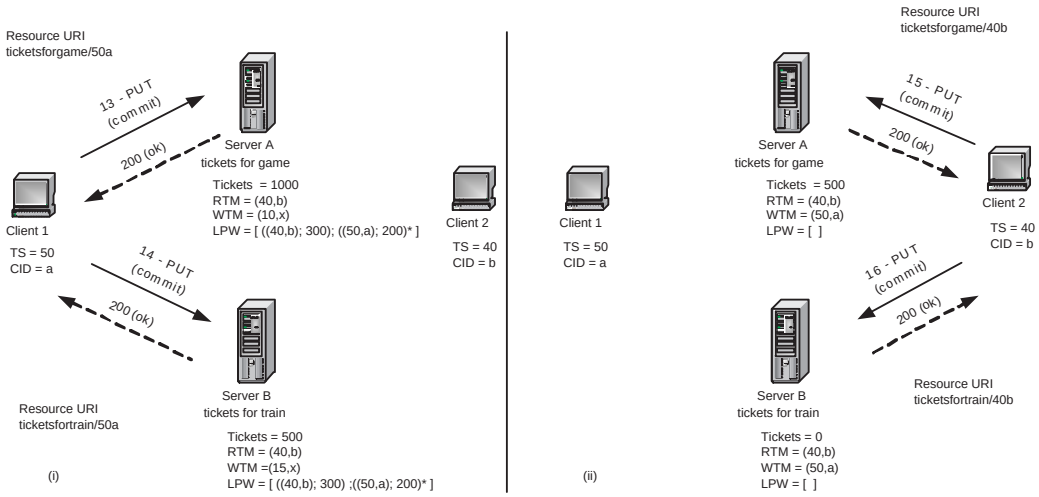

Fig. 3. (i) Client 1 commits its transaction. (ii) Client 2 commits its transaction.

when the transaction is over. The coordinator is responsible to guarantee the consistency, it decides to commit or abort the transaction based on the agents responses for the prewrite messages.

In our scheme, the isolation property is not respected, because clients can have access to the committed state (at WTM) and other possible intermediary states. However, as we showed the intermediary states may be reached (committed) or not. Two important considerations have to be made. First, there is an order of PWs to be processed, so the client may take into account this order in order to decide what to do. Second, there might be other PWs for the resource after the read access by the client. It is possible that after a read access with timestamp T, the LPW receives other PWs with earlier timestamps than T. Therefore the client with timestamp T in a read access obtains only a committed state before T and a possible and not complete sequence of updates.

Assuming the servers A and B store data item information using persistent database, durability is achieved at the end of the transaction.

Through the check of proposal feasibility, we verified that the number of transaction restarts of the TS2PC4RS is less than the restarts obtained by the BTS algorithm (Section 2.2.1). The TS2PC4RS also avoids buffering of read operations which forces clients to wait a certain time to receive information about a resource of interest.

The use of prewrite in the timestamp-based 2PC permits the abortion. The TS2PC4RS extends the TS2PC by allowing the agents to decide if a prewrite can be accepted or not. A condition of prewrite to be accepted is the resulting state must be consistent and the processing of all the earlier PWs keeps the states consistent. We also could reject a prewrite based on the agent's business rules, for instance, we can extend the TS2PC4RS algorithm to allow that the agent can reject requests for a large number of tickets if the number of remaining tickets is low.

Although we allow the transactions to read intermediary states, we do not use compensation [15, 16] in the TS2PC4RS. Compensation is used in cases where it is necessary to undo previously successfully completed work. As discussed in our optimistic approach [17], generally in order to use compensation it is important that compensating operations commute with transactions operations to avoid cascading aborts. We cannot reach commutativity in all applications domain where clients and services must interact. Our proposed algorithm can be used in cases where operations commutativity cannot be reached.

The proposed TS2PC4RS algorithm is deadlock free because the buffered prewrites are committed in timestamp order, so prewrites with larger timestamp wait for commitment of prewrites with lower

timestamps. Prewrites with lower timestamps do not wait for commitment of prewrites with larger timestamp.

In our proposal some mechanism of cleaning of prewrites may be necessary if the transactions take too long. In this situation, the updates of WTM may progress too slowly and LPWs may get too large. Some timeout mechanism may be used to address this problem. If a transaction does not commit its prewrites within a period of time then the agents may abort the corresponding prewrites. The agent has to have the permission from the coordinator beforehand.

4 TS2PC4RS and WS-Transaction

This section provides a comparison overview between the TS2PC4RS algorithm and the WS-Transaction specification which is part of the WS-* stack. WS-Transaction is based on three specifications: WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity.

WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity aim to provide transaction control implementation in WS-* web services, i.e., services based on SOAP. The services can be considered as activity oriented as they focus on action that might be performed rather than on the resources upon which they act. It is not possible to determine a uniform interface to the SOAP services. Each service can define its own interface based on the actions it has to perform. Thus, in order to promote a standard that actually implement the transaction control in this context, it is necessary to create specifications such as WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity.

WS-Coordination specification defines an extensible framework for coordinating activities using a coordinator and a set of coordination protocols. The framework enables participants to reach consistent agreement on the outcome of distributed activities.

WS-AtomicTransaction (WS-AT) [5] defines specific agreement coordination protocols for the Atomic Transaction coordination type. The protocols can be used when building applications that require consistent agreement on the outcome of short-lived distributed activities that have the all-or-nothing property, i.e., they ensure the atomicity property.

WS-BusinessActivity (WS-BA) [5] specification provides a more flexible protocol to reach the outcome. It is possible that participants reach different outcomes, in which some commit and some abort depending of the rules which participants assume in the transaction. WS-BA can be used when building applications require consistent agreement on the outcome of long-running distributed activities. Actions are applied immediately and are permanent, thus compensating actions are used in the event of an error. Compensation is used to recover from possible inconsistent states.

On the other hand, TS2PC4RS aims to provide a proposal to support transaction implementation in RESTful web services without adding complexity on the simplicity provided by the REST. Thus, TS2PC4RS applies to services that can be abstract as a *resource* and manipulated through a uniform interface. Our proposal is simpler because it does not require any specification that must be followed in order to implement transaction control in the RESTful web services domain. In our approach, the RESTful services have to be implemented according to the proposed algorithm and the clients and services involved in the transactions must agree with the information format used in the communication.

In TS2PC4RS algorithm, the client assumes the role of the coordinator and the client is responsible for the transaction context, so the client controls the sequence in which the services are invoked and the state of the transaction. It is assumed that the client does not need to distribute the transaction context information with the involved RESTful services, so it does not need a specification like WS-Coordination. Thus, for the simple cases of interaction, having the client as the coordinator leads to a faster and simpler implementation. The RESTful services are responsible to maintain the data items (the resources) they expose to third parties in a consistent state. TS2PC4RS assumes that the client knows all the RESTful services that the transaction needs to interact with and so it is not required any interfaces such as Activation or Registration service defined in WS-Coordination.

As TS2PC4RS can be used with short-lived and long-lived transactions, it is more flexible to deal with transactions in the web services domain. TS2PC4RS does not use compensation for long-lived

transaction, it softens the isolation property by using a non-locking algorithm and the *data item updated view concept*, described in Section 2.3.1, allowing the client decides what to do in cases where there are transactions in progress.

So TS2PC4RS and WS-Transaction are applied in different contexts. If the services can be abstracted as resources, TS2PC4RS is more appropriate. However, if the services are focused on actions that implement the business rules, WS-Transaction is more appropriate. The decision of which approach to use must be based on the business requirements involved, as well as in non-functional requirements that must be met.

5 Conclusions

Transactions implementation for RESTful web services using a timestamp based concurrency control and the two phase commit protocol is relatively simple and obeys the REST architectural style, which imposes some restrictions for implementation of web services. The clients has the responsibility for start, control, and commit or abort transactions. Clients assume the 2PC coordinator role. The REST resources have the responsibility to determine whether the prewrites are accepted based on the application business rules. They also ensure local consistency of its data items and the prewrites conversion into writes, in case the coordinator decides to commit the transaction, or prewrites discard, if the coordinator aborts the transaction. Therefore, a clearly separation of concerns between the client and server is achieved.

The example of purchase game and train tickets jointly allowed us to demonstrate the proposal feasibility. For the proposal, ROA is used, which is a concrete architecture to implement web services that conform to REST. The usage of timestamp concurrency control provides some advantages, for example, it is deadlock-free, as it does not use locks to information access control and so transactions are never blocked. The prewrites is not an actual lock, but it is equivalent to applying exclusive locks on data items for the time interval of the 2PC protocol.

The 2PC use also provides some advantages like its resilience to certain failures such as site failures and loss of messages. In order to achieve this resilience properly, no log information may be lost. So it is important to have a stable storage to record the logs on all agents and on the coordinator site, and that the 2PC timeouts are consistently set [11].

The possibility of a read operation to return not only the data item value, but also the WTM in which the value was last committed and a list of the prewrites prior to the read timestamp is one of the main advantages of the TS2PC4RS algorithm for REST services domain. This feature gives flexibility to the client to decide what should be made based on its application domain.

For future work, we intend to address the use of business process languages to model REST services transactions and how it can complement the use of protocols and algorithms like the one proposed in this work. The main idea is that transactions can be viewed as part of a business process, which in turn can be implemented by services composition. Another issue we intend to address is to propose solutions to other non-functional requirements in REST domain such as security and reliability.

References

1. W3C. Simple object access protocol (soap) 1.1, May 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.
2. W3C. Web services description language (wsdl) 1.1. Note, March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.
3. OASIS. Oasis web services security (wss) tc, February 2006. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`.
4. OASIS. Oasis web services reliable messaging (wsrm) tc, November 2004. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm`.
5. OASIS. Oasis web services transaction (ws-tx) tc, July 2007. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx`.

6. OASIS. Web services coordination (ws-coordination), July 2007. `http://docs.oasis-open.org/ws-tx/wscoor/2006/06`.
7. Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly & Associates, Sebastopol, California, May 2007.
8. R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, USA, 2000.
9. Roger L. Costello. Building web services the rest way, s.d. `http://www.xfront.com/REST-Web-Services.html` access date March 2008.
10. W3C. Naming and addressing: Uris, urls, ... `http://www.w3.org/Addressing/URL/uri-spec.html` access date March 2008.
11. S. Ceri and G. Pelagatti. *Distributed Databases, Principles and Systems*. McGraw-Hill, 1985.
12. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
13. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, concepts and design*. Addison-Wesley, 4th edition, 2005.
14. P. A. Bernstein and N. Goldman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.
15. J. Gray. The transaction concept: Virtues and limitations. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 144, Cannes, France, September 1981.
16. Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*, pages 95–106, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
17. L. A. Hiane S.Maciel and Celso M. Hirata. An optimistic technique for transactions control using REST architectural style. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 664–669. ACM, 2009.